Building Interpretable Models: From Bayesian Networks to Neural Networks

Building Interpretable Models: From Bayesian Networks to Neural Networks

A dissertation presented

by

Viktoriya Krakovna

to

The Department of Statistics

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Statistics

Harvard University

Cambridge, Massachusetts

September 2016

Thesis advisors: Prof. Jun Liu, Prof. Finale Doshi-Velez Viktoriya Krakovna

BUILDING INTERPRETABLE MODELS: FROM BAYESIAN NETWORKS TO NEURAL NETWORKS

ABSTRACT

This dissertation explores the design of interpretable models based on Bayesian networks, sum-product networks and neural networks.

As briefly discussed in Chapter 1, it is becoming increasingly important for machine learning methods to make predictions that are interpretable as well as accurate. In many practical applications, it is of interest which features and feature interactions are relevant to the prediction task. In Chapter 2, we develop a novel method, Selective Bayesian Forest Classifier (SBFC), that strikes a balance between predictive power and interpretability by simultaneously performing classification, feature selection, feature interaction detection and visualization. It builds parsimonious yet flexible models using tree-structured Bayesian networks, and samples an ensemble of such models using Markov chain Monte Carlo. We build in feature selection by dividing the trees into two groups according to their relevance to the outcome of interest. In Chapter 3, we show that SBFC performs competitively on classification and feature selection benchmarks in low and high dimensions, and includes a visualization tool that provides insight into relevant features and interactions. This is joint work with Prof. Jun Liu.

Sum-Product Networks (SPNs) are a class of expressive and interpretable hierarchical graphical models. In Chapter 4, we improve on LearnSPN, a standard structure learning algorithm for SPNs that uses hierarchical co-clustering to simultaneously identifying similar entities and similar features. The original LearnSPN algorithm assumes

that all the variables are discrete and there is no missing data. We introduce a practical, simplified version of LearnSPN, MiniSPN, that runs faster and can handle missing data and heterogeneous features common in real applications. We demonstrate the performance of MiniSPN on standard benchmark datasets and on two datasets from Google's Knowledge Graph exhibiting high missingness rates and a mix of discrete and continuous features. This is joint work with Moshe Looks (Google).

In Chapter 5, we turn our efforts from building interpretable models from the ground up to making neural networks more interpretable. As deep neural networks continue to revolutionize various application domains, there is increasing interest in making these powerful models more understandable, and narrowing down the causes of good and bad predictions. We focus on recurrent neural networks (RNNs), state of the art models in speech recognition and translation. Our approach to increasing interpretability is to combine an RNN with a hidden Markov model (HMM), a simpler and more transparent model. We explore different combinations of RNNs and HMMs: an HMM trained on LSTM states, and a hybrid model where an HMM is trained first, then a small LSTM is given HMM state distributions and trained to fill in gaps in the HMM's performance. We find that the LSTM and HMM learn complementary information about the features in the text, and we also apply the hybrid models to medical time series. This is joint work with Prof. Finale Doshi-Velez.

CONTENTS

ACKNOWLEDGEMENTS

During my meandering graduate school journey, I was fortunate to work with several brilliant researchers. I would like to thank Prof. Jun Liu for his feedback and insight, Prof. Finale Doshi-Velez for her insight, support, and inspiration, and Dr. Moshe Looks for guidance and encouragement during my internship at Google.

I am grateful to my comrades at the statistics and computer science departments for their good humor and curiosity, and the many interesting conversations that taught me a great deal. I would especially like to thank David Duvenaud, Ryan Adams, Finale and others at the machine learning lab for welcoming me to the computer science department in my last year; and Yang Chen for her perpetual encouragement on our shared path through graduate school.

I am very thankful to my housemates at Citadel House for their moral support, listening to my research-related rants, and helping to debug my code. I am also grateful to my co-founders and teammates at the Future of Life Institute for their encouragement and advice, as well as making my graduate school years more interesting and rewarding.

I am grateful to my parents and sisters for their faith in me, and the scientists in my family who have inspired me as role models. Lastly, I am deeply thankful to my husband, János Kramár, for proofreading all my work, and for being a great source of understanding, validation and patience for the past 11 years.

1

INTERPRETABILITY - WHY AND HOW?

My graduate work has taken place in the intersection of statistics and machine learning, and I have sought to bring together ideas and values from both disciplines. While statisticians emphasize the insight and elegance of models, machine learning researchers tend to focus on predictive power. Model interpretability has received a lot of thought in the statistics community, while the machine learning community has only recently started taking it seriously. In both fields, it is commonly believed that there is a tradeoff between interpretability and predictive power. While this tradeoff is real, it is often possible to make a model more interpretable without significantly impairing performance. In this dissertation, I strive to build models that are both powerful and interpretable.

Why is interpretability important? We want the models we build to be useful and reliable in a variety of applications. As machine learning is increasingly applied outside low-risk domains like games (Mnih et al., 2013) and image labeling (Vinyals et al., 2015), it becomes more and more important for the methods to be understandable so that they can be trusted. For example, recent European Union regulations underscore the importance of interpretability of automated decision-making, mandating a "right to explanation" where a user can ask for an explanation of an automatic decision (Goodman and Flaxman, 2016). In high-stakes applications, such as medicine, the users care about the robustness and transparency of a method as much, if not more, than about its accuracy. Machine learning methods can be error-prone and difficult to debug (Sculley et al., 2014), and make very different errors than humans do (Papernot et al., 2016). As they become more autonomous and powerful, avoiding accidents and

unintended consequences will be a crucial and difficult challenge (Amodei et al., 2016). This makes it all the more important to understand the weaknesses of our models and be able to track down the causes of poor predictions.

As interpretability is based on human understanding, it is an intrinsically subjective and multifaceted quality, rather than a single axis. A model or algorithm can be considered intelligible to humans in multiple ways, falling under the broad categories of transparency and post-hoc interpretability (Lipton, 2016). While works such as (Ribeiro et al., 2016) and (Turner, 2016) develop post-hoc explanations for black-box models, I focus on transparency, specifically model parsimony and the ability to trace back from a prediction or model component to particularly influential features in the data, similarly to Kim et al. (2015).

In this work, I explore several fundamentally different approaches to building interpretable models. I start by building a novel Bayesian method with interpretability in mind from the ground up. SBFC produces good predictions while identifying and visualizing the features and interactions that most strongly drive these predictions. While building an interpretable model from scratch is the most reliable way to ensure interpretability (Rüping, 2006), the resulting method will not necessarily be adopted by practitioners over more popular opaque methods.

I thus proceed to improve upon two existing well-known methods, in the domains of sum-product networks and neural networks. The SPN model is already interpretable, and my contribution is to simplify a classical learning algorithm and make it more broadly applicable. On the other hand, deep neural networks are famously opaque and widely used. While there has been some work on interpreting and visualizing convolutional networks (Yosinski et al., 2015), little has been done to increase the interpretability of recurrent networks. I attempt to bridge this gap by combining recurrent networks with more transparent models, and visualizing how these models complement each other in terms of features they learn.

2

SELECTIVE BAYESIAN FOREST CLASSIFIER: SIMULTANEOUS FEATURE SELECTION, INTERACTION DETECTION, AND CLASSIFICATION

2.1 INTRODUCTION

Feature selection and classification are key objectives in machine learning. Many approaches have been developed for these two problems, usually tackling them separately. However, performing classification on its own tends to produce black box solutions that are difficult to interpret, while performing feature selection alone can be difficult to justify without being validated by prediction. In addition to screening for relevant features, it is also useful to detect interactions between them, and this problem becomes especially difficult in high dimensions. In many decision support systems, e.g. in medical diagnostics, the users care about which features and feature interactions contributed to a particular decision.

Many methods focus either on feature selection or on identifying feature interactions, or deal with these tasks in two independent steps. However, feature selection and feature interaction detection are often closely related. Without identifying feature interactions, feature selection can easily omit features that have weak marginal influence on the class label individually but have a large influence on it jointly. Conversely, without feature selection, identification of feature interactions tends to be computationally infeasible. Therefore, it is a good idea to accomplish these two tasks simultaneously. Selective Bayesian Forest Classifier (SBFC) combines predictive power and interpretabil-

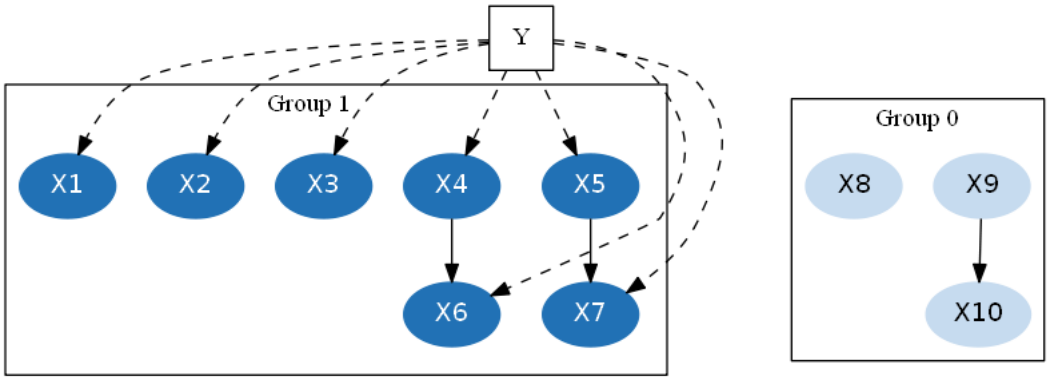


Figure 1: Example of a SBFC graph

ity, by performing classification, feature selection, and feature interaction detection at the same time. Our method also provides a visual representation of the features and feature interactions that are relevant to the outcome of interest.

The main idea of SBFC is to construct an ensemble of Bayesian networks (Pearl, 1988), each constrained to a forest of trees divided into signal and noise groups based on their relationship with the class label Y (see Figure 1 for an example). The nodes and edges in Group 1 represent relevant features and interactions. SBFC is inspired by Naïve Bayes, an exceedingly simple yet surprisingly effective classifier, that assumes independence between the features conditional on the class label. Starting from the Naïve Bayes framework, we build dependence structures on the features. The features are partitioned into two groups based on their relationships with the class label, and the groups are further divided into independent subgroups, with each subgroup modeled by a tree structure. Such models are easy to sample using Markov chain Monte Carlo (MCMC). We combine their predictions using Bayesian model averaging, and aggregate their feature and interaction selection.

We show that SBFC performs competitively with state-of-the-art methods on 25 low-dimensional and 6 high-dimensional benchmark data sets. By adding noise features to a synthetic data set, we compare feature selection and interaction detection performance as the signal to noise ratio decreases (Figure 4). We use a high-dimensional data set from

the NIPS 2003 feature selection challenge to demonstrate SBFC's superior performance on a difficult feature selection task (Figure 6), and illustrate the visualization tool on a heart disease data set with meaningful features (Figure 5). SBFC is a good choice of algorithm for applications where interpretability matters along with predictive power.

2.2 RELATED WORK

2.2.1 *Naïve Bayes*

The Naïve Bayes classifier (NB) (Duda and Hart, 1973) learns from training data the conditional probability of each feature in $\boldsymbol{X} = (X_1, \ldots, X_d)$, given class label Y. It then classifies a new instance into the most probable class, assuming that the features are conditionally independent given the class:

$$\begin{aligned} &P(Y = y | X_1 = x_1, \ldots, X_d = x_d) \\ \propto &P(Y = y) \prod_{j=1}^{d} P(X_j = x_j \mid Y = y), \end{aligned}$$

where the distribution $P(X_j = x_j \mid Y = y)$ is estimated from the data. The assumption of conditional independence among all the features is far from realistic, but the performance of the NB classifier has been surprisingly good, competitive with state-of-art classifiers in many real applications (Zhang, 2004). A possible explanation for NB's good performance is that it avoids overfitting by modeling the distribution of $\mathbf{X}$ conditional on Y, instead of directly focusing on a predictive model of Y on $\mathbf{X}$. Although the conditional independence distribution $\prod_{i=1}^{d} P(X_j \mid Y)$ might differ significantly from the true probability distribution $P(\boldsymbol{X} \mid Y)$, their overlap is often good enough for classification, which is based on a 0-1 loss function (Domingos and Pazzani, 1997).

2.2.2 *Bayesian Network model*

The restrictive conditional independence assumption of NB often harms its performance when features are correlated. At the opposite extreme, we have the unrestricted Bayesian network model (Pearl, 1988). A Bayesian network (BN) is a directed acyclic graph that encodes a joint probability distribution over $\boldsymbol{X}$. It contains two components: G and Θ, where G represents the directed acyclic graph, and Θ stands for the parameters needed to describe the joint probability distribution. Each node represents a feature, and a directed edge corresponds to a "parent $\rightarrow$ child" dependence relationship between the features, so that each feature X_j is independent of its non-descendants given its parents Λ_j.

The probability distribution over $\mathbf{X}$ can be written as

$$P(\mathbf{X}) = \prod_{j=1}^{d} P(X_j|\Lambda_j).$$

Bayesian network models present a tremendous computational challenge. Structure learning is NP-hard in the general case (Heckerman et al., 1995), as is exact inference (Cooper, 1990). The flexibility of the BN model is also its curse when the number of features is large, and the network structure can be difficult to intepret.

2.2.3 *Tree-structured Bayesian methods*

Tree structures are frequently used in computer science and statistics, because they provide adequate flexibility to model complex structures, yet are constrained enough to facilitate computation. SBFC was inspired by tree-based methods such as Tree-Augmented Naïve Bayes (TAN) (Friedman et al., 1997), Averaged One-Dependence Estimators (AODE) (Webb et al., 2005), and Hidden Naïve Bayes (HNB) (Zhang et al.,

2005), which relax the conditional independence assumption of NB to allow tree structures on the features.

TAN finds the optimal tree on all the features using the minimum spanning tree algorithm, with the class label Y as a second parent for all the features. The class label Y is the first parent of all the features, and each feature has another feature as its second parent, except for the root vertex in G, which has only one parent Y. While the search for the best unrestricted Bayesian network is usually an intractable task (Heckerman et al., 1995), the computational complexity of TAN is only $O(d^2 n)$, where d is the number of features and n is the sample size (Chow and Liu, 1968).

AODE constrains the model structure to a tree where all the features are children of the root feature, with Y as a second parent, and uses model averaging over model with all possible root features.

$$P(Y, \mathbf{X}) = P(Y, X_k) \prod_{j \neq k} P(X_j \mid Y, X_k).$$

HNB, an extension of AODE, designates a hidden parent X_{p_j} for each feature X_j, and assumes that the impact of this hidden parent on X_j is a weighted average of the impact of all the other features on X_j:

$$P(X_j \mid X_{p_j}, Y) = \sum_{k \neq j} w_{jk} P(X_j | X_k, Y), \ \sum_{k \neq j} w_{jk} = 1.$$

These methods put all the features into a single tree, which can be difficult to interpret, especially for high-dimensional data sets. We extend these methods by building forests instead of single-tree graphs, and introducing selection of relevant features and interactions.

2.2.4 *Adding feature selection*

While the above approaches focus on building a dependence structure, the following methods augment Naïve Bayes with feature selection. Selective Bayesian Classifier (SBC) (Langley and Sage, 1994) applies a forward greedy search method to select a subset of features to construct a Naïve Bayes model, while Evolutional Naïve Bayes (ENB) (Jiang et al., 2005) uses a genetic algorithm with the classification accuracy as its fitness function.

More generally, one can use feature selection as a preprocessing step for any classification algorithm. Wrapper methods (Kohavi and John, 1997) select a subset of features tailored for a specific classifier, treating it as a black box. feature Selection for Clustering and Classification (VSCC) (Andrews and McNicholas, 2014) searches for a feature subset that simultaneously minimizes the within-class variance and maximizes the between-class variance, and remains efficient in high dimensions. Categorical Adaptive Tube Covariate Hunting (CATCH) (Tang et al., 2014) selects features based on a nonparametric measure of the relational strength between the feature and the class label.

Our approach, however, is to integrate feature selection into the classification algorithm itself, allowing it to influence the models built for classification. A classical example is Lasso (Tibshirani, 1996), which performs feature selection using L_1 regularization. Some decision tree classifiers, like Random Forest (Breiman, 2001) and BART (Chipman et al., 2010), provide importance measures for features and the option to drop the least significant features. In many applications, it is also key to identify relevant feature interactions, such as epistatic effects in genetics. Interaction detection methods for gene association models include Graphical Gaussian models (Andrei and Kendziorski, 2009) and Bayesian Epistasis Association Mapping (BEAM) (Zhang and Liu, 2007). BEAM introduces a latent indicator that partitions the features into several

groups based on their relationship with the class label. One of the groups in BEAM is designed to capture relevant feature interactions, but is only able to tractably model a small number of them. SBFC extends this framework, using tree structures to represent an unlimited number of relevant feature interactions.

Our work is similar to the Extended TAN algorithm (ETAN) (de Campos et al., 2014), an extension of TAN that allows it to have forest structure and features disconnected from the class label. While ETAN uses a variation on the Edmonds algorithm for finding the minimum spanning forest, we learn a collection of forest structures using MCMC.

2.3 SELECTIVE BAYESIAN FOREST CLASSIFIER (SBFC)

SBFC combines feature selection and structure building, partitioning the features based on their relation to the class label, and building tree structures within the partitions. Then it uses MCMC to sample from the space of these graph structures, and performs classification based on multiple sampled graphs via Bayesian model averaging.

2.3.1 *Model*

Given n observations with class label Y and d discrete features X_j, $j = 1, \ldots, d$, we divide the features into two groups based on their relation to Y (see Figure 1 for an example):

GROUP 0 (NOISE): features that are unrelated to Y

GROUP 1 (SIGNAL): features that are related to Y

We further partition each group into non-overlapping subgroups mutually independent of each other conditional on Y. The number of subgroups in each group is

unknown in our model - we sample it using MCMC instead of assigning a fixed number a priori. For each subgroup, we infer a tree structure describing the dependence relationships between the features (many subgroups will consist of one node and thus have a trivial dependence structure). Note that we model the structure in the noise group as well as the signal group, since an independence assumption for the noise features could result in correlated noise features being misclassified as signal features.

The overall dependence structure is thus modeled as a forest of trees, representing conditional dependencies between the features (no causal relationships are inferred). The class label Y is a parent of every feature in Group 1 (edges to Y are omitted in subsequent figures). We will refer to the combination of a group partition and a forest structure as a graph.

The prior consists of a penalty on the number of edges between features in each group and a penalty on the number of signal nodes (i.e., edges between features and Y)

$$P(G) \propto d^{-\lambda_x \cdot (E_0(G) + E_1(G)/v) - \lambda_y \cdot D_1(G)/v}$$

where $D_i(G)$ is the number of nodes and $E_i(G)$ is the number of edges in Group i of graph G, while v is a constant equal to the number of classes.

The prior scales with the number of features d to penalize very large, hard-to-interpret trees in high dimensional cases. The terms corresponding to the signal group are divided by the number of possible classes v, to avoid penalizing large trees in the signal group more than in the noise group by default. The penalty coefficients in the prior, $\lambda_x = 4$ and $\lambda_y = 1$, were determined empirically to provide good classification and feature selection performance (there is a relatively wide range of coefficients that produce similar results).

Table 1: Parent sets for each feature type

Type of feature X_j	Parent set Λ_j
Group 0 root	$\emptyset$
Group 0 non-root	$\{X_{p_j}\}$
Group 1 root	$\{Y\}$
Group 1 non-root	$\{Y, X_{p_j}\}$

Given the training data $X_{(n\times d)}$ (with columns $\boldsymbol{X}_j$, $j = 1, \ldots, d$) and $\boldsymbol{y}_{(n\times 1)}$, we break down the graph likelihood according to the tree structure:

$$\begin{aligned} P(X, \boldsymbol{y}|G) &= P(\boldsymbol{y}|G)P(X|\boldsymbol{y}, G) \\ &= P(\boldsymbol{y}) \prod_{j=1}^{d} P(\boldsymbol{X}_j|\boldsymbol{\Lambda}_j) \end{aligned}$$

Here, Λ_j is the set of parents of X_j in graph G. This set includes the parent X_{p_j} of X_j unless X_j is a root, and Y if X_j is in Group 1, as shown in Table 1. We assume that the distributions of the class label Y and the graph structure G are independent a priori.

Let v_j and w_j be the number of possible values for X_j and Λ_j respectively. Then our hierarchical model for X_j is

$$\begin{aligned} [X_j|\Lambda_j = \Lambda_{jl}, \boldsymbol{\Theta}_{jl} = \boldsymbol{\theta}_{jl}] &\sim \text{Mult}(\boldsymbol{\theta}_{jl}),\ l = 1, \ldots, w_j \\ \boldsymbol{\Theta}_{jl} &\sim \text{Dirichlet}\left(\frac{\alpha}{w_j v_j}\mathbf{1}_{v_j}\right) \end{aligned}$$

Each conditional Multinomial model has a different parameter vector $\boldsymbol{\Theta}_{jl}$. We consider the Dirichlet hyperparameters to represent "pseudo-counts" in each conditional model (Friedman et al., 1997). Let n_{jkl} be the number of observations in the training data with $X_j = x_{jk}$ and $\Lambda_j = \Lambda_{jl}$, and $n_{jl} = \sum_{k=1}^{v_j} n_{jkl}$. Then

$$P(\boldsymbol{X}_j|\boldsymbol{\Lambda}_j, \boldsymbol{\Theta}_{j1}, \ldots, \boldsymbol{\Theta}_{jw_j}) = \prod_{l=1}^{w_j} \prod_{k=1}^{v_j} \theta_{jkl}^{n_{jkl}}$$

We then integrate out the nuisance parameters $\mathbf{\Theta}_{jl}$, $l = 1, \ldots, w_j$. The resulting likelihood depends only on the hyperparameter α and the counts of observations for each combination of values of X_j and Λ_j.

$$P(\boldsymbol{X}_j|\mathbf{\Lambda}_j) = \prod_{l=1}^{w_j} \frac{\Gamma\left(\frac{\alpha}{w_j}\right)}{\Gamma\left(\frac{\alpha}{w_j} + n_{jl}\right)} \prod_{k=1}^{v_j} \frac{\Gamma\left(\frac{\alpha}{w_j v_j} + n_{jkl}\right)}{\Gamma\left(\frac{\alpha}{w_j v_j}\right)}$$

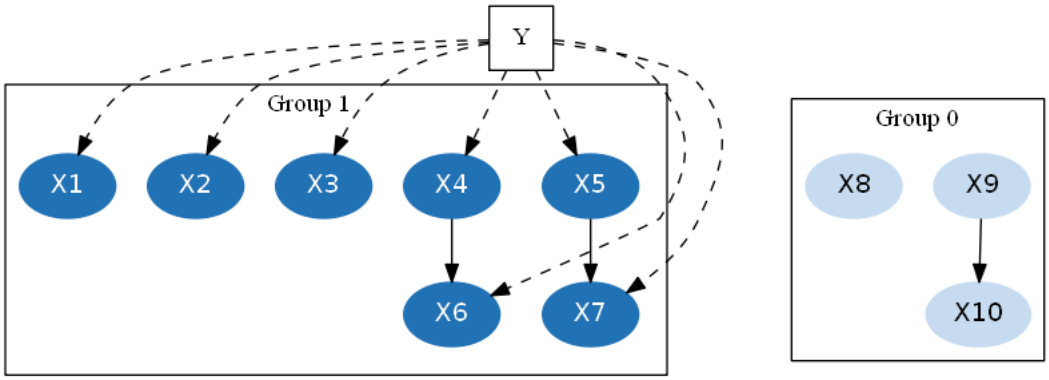
This is the Bayesian Dirichlet score, which satisfies likelihood equivalence (Heckerman et al., 1995). Namely, reparametrizations of the model that do not affect the conditional independence relationships between the features, for example by pivoting a tree to a different root, do not change the likelihood.

2.3.2 *MCMC Updates*

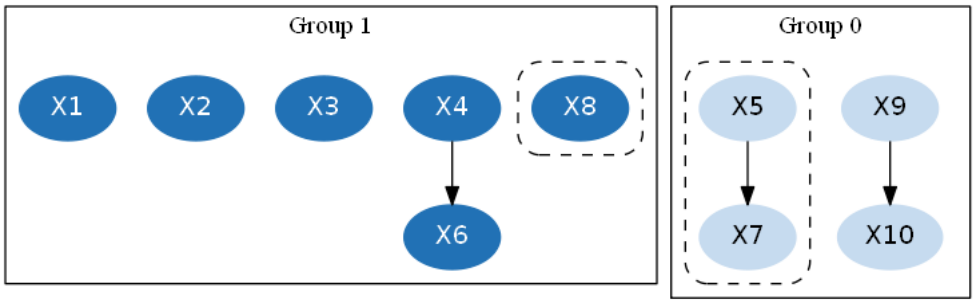
SWITCH TREES: Randomly choose trees $T_1, \ldots, T_k$ without replacement, and propose switching each tree to the opposite group one by one (see Figure 2b). We chose $k = 10$ for computational efficiency reasons, as higher values such as $k = 20$ did not improve performance or mixing (though it could be useful to scale the value of k with the number of variables). This is a repeated Metropolis update.

REASSIGN SUBTREE: Randomly choose a node X_j, detach the subtree rooted at this node and choose a different parent node for this subtree (see Figure 2c). This is a Gibbs update, so it is always accepted.

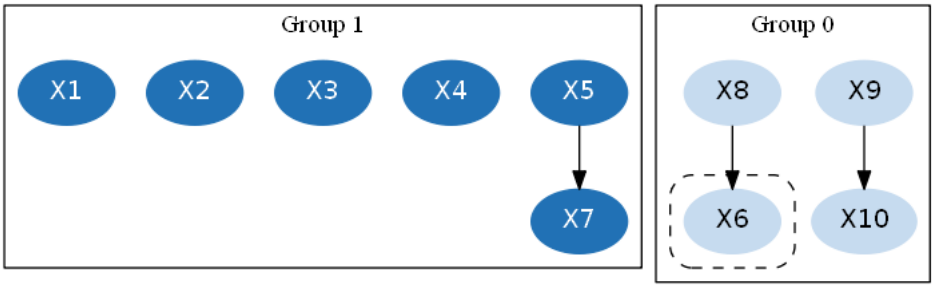
We consider the set of nodes $X_{j'}$ that are not descendants of X_j as candidate parent nodes (to avoid creating a cycle), with corresponding graphs $G_{j'}$. We also consider a "null parent" option for each group, where X_j becomes a root in that group, with corresponding graph $\tilde{G}_i$ for group i. Choose a graph G^* from this set according to the conditional posterior distribution $\pi(G^*)$ (conditioning on the parents of all



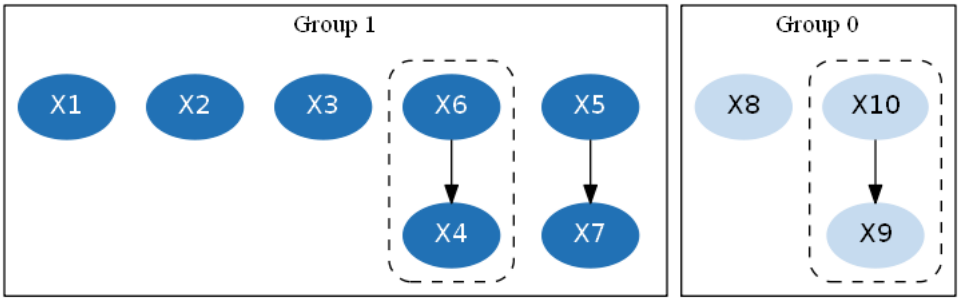
(a) Original graph



(b) Switch Trees: switch tree $\{X_5, X_7\}$ to Group 0, switch tree $\{X_8\}$ to Group 1]



(c) Reassign Subtree: reassign node X_6 to be a child of node X_8



(d) Pivot Trees: nodes X_6 and X_{10} become tree roots

Figure 2: Example MCMC updates applied to the graph in Figure 2a

the nodes except X_j, and on the group membership of all the nodes outside the subtree). The subtree joins the group of its new parent.

As a special case, this results in a tree merge if X_j was a root node, or a tree split if X_j becomes a root (i.e. the new parent is null). Note that the new parent can be the original parent, in which case the graph does not change.

PIVOT TREES: Pivot all the trees by randomly choosing a new root for each tree (see Figure 2d). By likelihood equivalence, this update is always accepted.

For computational efficiency, in practice we don't pivot all the trees at each iteration. Instead, we just pivot the tree containing the chosen node X_j within each Reassign Subtree move, since this is the only time the parametrization of a tree matters. This implementation produces an equivalent sampling mechanism.

2.3.3 *Detailed balance for MCMC updates*

Let $\pi(G) = P(G|X, y)$ be the target distribution.

Switch Trees

Let p be the probability that tree T is among the k trees proposed for switching - this only depends on k and the number of trees in the graph, which is constant during the Switch update. Let G be the starting graph when proposing to switch T, and G^* be the graph obtained from G by switching T. Then,

$$\begin{aligned}
\pi(G)P(G \to G^*) &= \pi(G)P(G \to G^* \text{ proposed})P(G \to G^* \text{ accepted}|G \to G^* \text{ proposed}) \\
&= \pi(G) \cdot p \cdot \min\left(1, \frac{\pi(G^*)p}{\pi(G)p}\right) \\
&= p \min\left(\pi(G), \pi(G^*)\right)
\end{aligned}$$

Since this expression is symmetric in G and G^*, we can conclude that the detailed balance condition holds:

$$\pi(G)P(G \to G^*) = \pi(G^*)P(G \to G).$$

Reassign Subtree

Let G be the starting graph. We pick any node X, and choose a non-descendant (or null) node to be its parent. Suppose X has c descendants - then the number of possible resulting graphs G_i is $d - c$ (including G itself). The backtracking update from G^* is reassigning X again, which results in the same possible graphs G_i.

$$\begin{aligned}
\pi(G)P(G \to G^*) &= \pi(G)P(G \to G^* \text{ proposed}) \\
&= \pi(G)\frac{1}{d}\frac{\pi(G^*)}{\sum_{i=1}^{d-c}\pi(G_i)} \\
&= \pi(G^*)\frac{1}{d}\frac{\pi(G)}{\sum_{i=1}^{d-c}\pi(G_i)} \\
&= \pi(G^*)P(G^* \to G)
\end{aligned}$$

Pivot Trees

Let G be the starting graph when pivoting tree T of size $|T|$. Let G^* be the graph obtained by pivoting T. By likelihood equivalence, $\pi(G) = \pi(G^*)$, so the proposal is always accepted.

$$\begin{aligned}\pi(G)P(G \to G^*) &= \pi(G)P(G \to G^* \text{ proposed}) \\ &= \pi(G)\frac{1}{|T|} \\ &= \pi(G^*)\frac{1}{|T|} \\ &= \pi(G^*)P(G^* \to G)\end{aligned}$$

2.3.4 *Classification Using Bayesian Model Averaging*

Graphs are sampled from the posterior distribution using the MCMC algorithm. We apply Bayesian model averaging (Hoeting et al., 1998) rather than using the posterior mode for classification. For each possible class, we average the probabilities over a thinned subset of the sampled graph structures, and then choose the class label with the highest average probability. Given a test data point $\boldsymbol{x}^{\text{test}}$, we find

$$\begin{aligned}&P(Y = y|\boldsymbol{X} = \boldsymbol{x}^{\text{test}}, X, \boldsymbol{y}) \\ \propto &\sum_{i=1}^{S} P(Y = y|\boldsymbol{X} = \boldsymbol{x}^{\text{test}}, G_i)P(G_i|X, \boldsymbol{y})\end{aligned}$$

where S is the number of graphs sampled by MCMC (after thinning by a factor of 50). We use training data counts to compute the posterior probability of the class label given each sampled graph G_i.

Let R_1 be the index set of root nodes in Group 1 in graph G. Let E_1 be the index set of non-root ("edge") nodes in Group 1 in G. Let p_i be the index of the parent of covariate x_i.

We have a new data point $\mathbf{x}^{new}$, and for classification we need the posterior probability that $y = y^{new}$ for every value y^{new}. Let the training data be $X, \mathbf{y}$, with sample size

N. We approximate the probability $P(\mathbf{x} = \mathbf{x}^{new}|y = y^{new}, G, X, \mathbf{y})$ using counts from training data, where $\#(A)$ is the count of training data points satisfying condition A. Thanks to the tree structure of G, the only features involved in each count are x_i, the parent of x_i (if it exists), and y.

$$
\begin{aligned}
&P(y = y^{new}|\mathbf{x} = \mathbf{x}^{new}, G, X, \mathbf{y}) \\
\propto &P(y = y^{new}|X, \mathbf{y})P(\mathbf{x} = \mathbf{x}^{new}|y = y^{new}, G, X, \mathbf{y}) \\
\approx &\frac{\#(y = y^{new})}{N} \prod_{i \in R_1} P(x_i = x_i^{new}|y = y^{new}, X, \mathbf{y}) \\
&\times \prod_{i \in E_1} P(x_i = x_i^{new}|x_{p_i} = x_{p_i}^{new}, y = y^{new}, X, \mathbf{y}) \\
\approx &\frac{\#(y = y^{new})}{N} \prod_{i \in R_1} \frac{\#(x_i = x_i^{new}, y = y^{new})}{\#(y = y^{new})} \\
&\times \prod_{i \in E_1} \frac{\#(x_i = x_i^{new}, x_{p_i} = x_{p_i}^{new}, y = y^{new})}{\#(x_{p_i} = x_{p_i}^{new}, y = y^{new})} \\
\approx &\frac{\#(y = y^{new})}{N} \prod_{i \in R_1} \frac{\#(x_i = x_i^{new}, y = y^{new}) + \frac{1}{N}\#(x_i = x_i^{new})}{\#(y = y^{new}) + 1} \\
&\times \prod_{i \in E_1} \frac{\#(x_i = x_i^{new}, x_{p_i} = x_{p_i}^{new}, y = y^{new}) + \frac{1}{N}\#(x_i = x_i^{new})}{\#(x_{p_i} = x_{p_i}^{new}, y = y^{new}) + 1}
\end{aligned}
$$

Note that instead of using raw counts in the numerator and denominator, we add 1 in the denominator and $\frac{1}{N}\#(x_i = x_i^{new})$ in the numerator. This increases the robustness of this approximation of the probabilities by avoiding zero values, and ensures that the probabilities still add up to 1 (since $\frac{1}{N}\sum\#(x_i = x_i^{new}) = \frac{1}{N}N = 1$).

Now, we find these classification probabilities for each sampled graph G, and choose the value y^{new} that maximizes the *weighted* average probability

$$P(y = y^{new}|\mathbf{x} = \mathbf{x}^{new}, X, \mathbf{y}) = \sum_G P(G|X, \mathbf{y})P(y = y^{new}|\mathbf{x} = \mathbf{x}^{new}, G, X, \mathbf{y})$$

3

INTERPRETABLE SELECTION AND VISUALIZATION OF FEATURES AND INTERACTIONS USING BAYESIAN FORESTS: SBFC IN PRACTICE

3.1 EXPERIMENTS ON REAL DATA

We compare our classification performance with the following methods.

BART: Bayesian Additive Regression Trees, R package `BayesTree` (Chipman et al., 2010),

C5.0: R package `C50` (Quinlan, 1993),

CART: Classification and Regression Trees, R package `tree` (Breiman et al., 1984),

LASSO: R package `glmnet` (Friedman et al., 2010),

LR: logistic regression,

NB: Naïve Bayes, R package `e1071` (Duda and Hart, 1973)

RF: Random Forest, R package `ranger` (Breiman, 2001),

SVM: Support Vector Machines, R package `e1071` (Evgeniou et al., 2000),

TAN: Tree-Augmented Naïve Bayes, R package `bnlearn` (Friedman et al., 1997).

We use 25 small benchmark data sets used by Friedman et al. (1997) and 6 high-dimensional data sets (Guyon et al., 2005), all from the UCI repository (Lichman, 2013),

described in Table 3. We split the large data sets into a training set and a test set, and use 5-fold cross validation for the smaller data sets (we try both approaches for the high-dimensional `arcene` data set). We remove the instances with missing values, and discretize continuous features, using Minimum Description Length Partitioning (Fayyad and Irani, 1993) for the small data sets and binary binning (Dougherty et al., 1995) for the large ones. For a data set with d features, we run SBFC for $\max(10000, 10d)$ iterations, which has empirically been sufficient for stabilization. Figure 3 compares SBFC's classification performance to the other methods.

We evaluate SBFC's feature ranking and interaction detection performance on the data sets `heart`, `corral`, and `madelon`, in Figures 5, 4, and 6 respectively. We also compare feature selection in Figures 7 and 8. We compare SBFC's feature selection performance to Lasso, as well as RF's `importance` metric and BART's `varcount` metric, which rank features by their influence on classification, in Figures 5c, 4e, 4f, and 6c. We illustrate the structures learned by SBFC on these data sets using sampled graphs, shown in Figures 5a, 4a, 4b, and 6a, and average graphs over all the MCMC samples, shown in Figures 5b, 4c, 4d, and 6b.

In the average graphs, the nodes are color-coded according to relevance, based on the proportion of sampled graphs where the corresponding feature appeared in Group 1 (dark-shaded nodes appear more often). Edge thickness also corresponds to relevance, based on the proportion of samples where the corresponding feature interaction appeared. To avoid clutter, only edges that appear in at least 10% of the sampled graphs are shown, and nodes that appear in Group 0 more than 80% of the time are omitted for high-dimensional data sets. Average graphs are undirected and do not necessarily have a tree structure. They provide an interpretable visual summary of the relevant features and feature interactions.

Runtimes on the high-dimensional data sets are shown in Table 2 (on an AMD Opteron 6300-series processor). As shown in Table 2, the runtime of SBFC scales

Table 2: Average runtime comparison (in seconds) on the high-dimensional data sets.

Data Set	BART	C5.0	CART	Lasso	LR	NB	RF	SBFC	SVM	TAN
ad	370	9	6	133	676	44	31	290	82	140
arcene	31	17	15	4	40	61	16	3600	40	n/a
arcene-cv	272	164	63	48	210	193	87	3900	120	n/a
gisette	834	98	18	740	4600	118	69	8100	1000	n/a
isolet	n/a	9	3	n/a	n/a	31	37	1400	200	24
madelon	231	4	1	26	13	7	8	60	32	9
microsoft	3380	44	4.7	110	92	52	452	140	290	12

approximately as $d \cdot n \cdot 2 \cdot 10^{-4}$ seconds, where d is the number of features and n is the number of instances, and SBFC takes somewhat longer to run than many of the other methods on high-dimensional data sets. SBFC's memory usage scales quadratically with d.

We examine MCMC diagnostics for some of the data sets in Figure 9, which shows the log posterior trace plot for a single run on each data set for 10000 iterations, with burn-in for 1/5 (2000) of the iterations (in general the default number of iterations is $\min(10000, 10d)$). It's interesting how the trace plot changes as we add features to the corral data set: the log posterior is reaching an upper limit for the original data with 6 features, but not for 100 or more features, and the trace plot for 1000 features shows worse mixing. This effect generally holds when comparing data sets with a low number of features, like australian or heart (13-14 features), with data sets with a high number of features, like madelon (500 features). The most problematic-looking plot is for the chess data, where we observe a large jump in the log posterior at around 7000 iterations, well past the burn-in cutoff of 2000. This suggests that we should run the algorithm for longer than 10000 iterations.

Table 3: Data set properties (Friedman et al., 1997)

Data set	Features	Classes	Instances	
			Train	Test
australian	14	2	690	CV-5
breast	10	2	683	CV-5
chess	36	2	2130	1066
cleve	13	2	296	CV-5
corral	6	2	128	CV-5
crx	15	2	653	CV-5
diabetes	8	2	768	CV-5
flare	10	2	1066	CV-5
german	20	2	1000	CV-5
glass	9	6	214	CV-5
glass2	9	2	163	CV-5
heart	13	2	270	CV-5
hepatitis	19	2	80	CV-5
iris	4	3	150	CV-5
letter	16	26	15000	5000
lymphography	18	4	148	CV-5
mofn-3-7-10	10	2	300	1024
pima	8	2	768	CV-5
satimage	36	6	4435	2000
segment	19	7	1540	770
shuttle-small	9	6	3866	1934
soybean-large	35	19	562	CV-5
vehicle	18	4	846	CV-5
vote	16	2	435	CV-5
waveform-21	21	3	300	4700
ad	1558	2	2276	988
arcene	10000	2	100	100
arcene-cv	10000	2	200	CV-5
gisette	5000	2	6000	1000
isolet	617	26	6238	1559
madelon	500	2	2000	600
microsoft	294	2	32711	5000

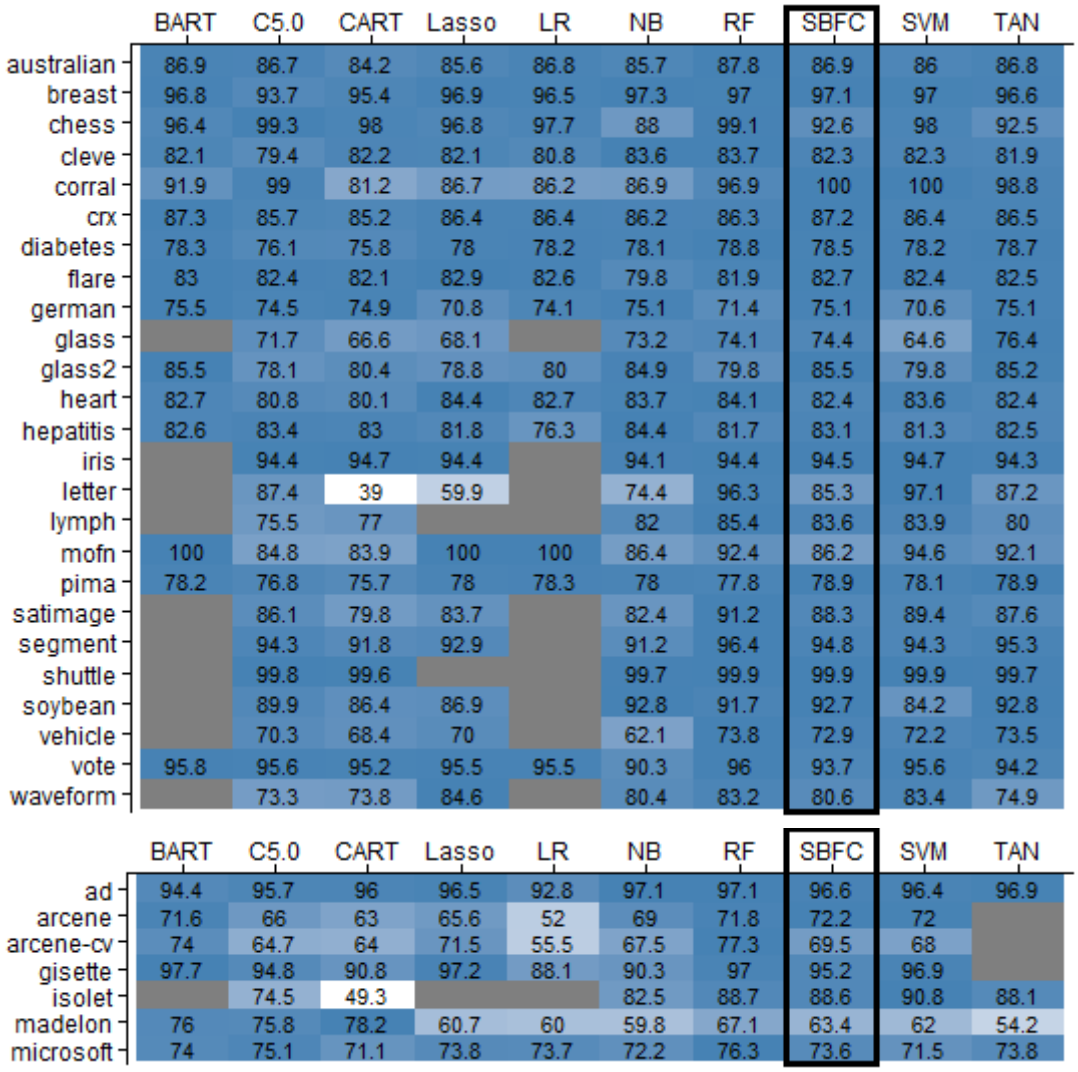
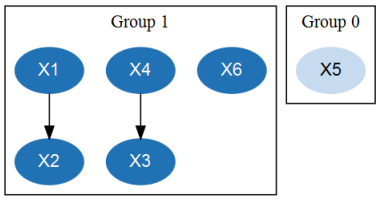
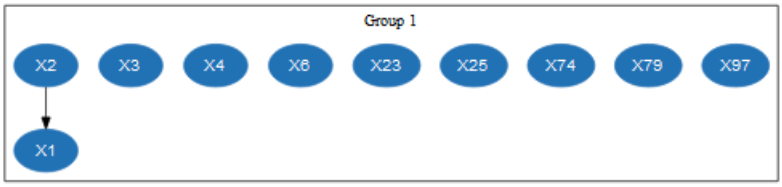


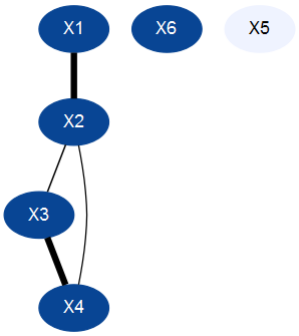
Figure 3: Classification accuracy on low- and high-dimensional data sets, showing average accuracy over 5 runs for each method, with the top half of the methods in bold for each data set. Note that some of the classifiers could not handle multiclass data sets, and TAN timed out on the highest-dimensional data sets. SBFC performs competitively with SVM, TAN and some decision tree methods (BART and RF), and generally outperforms the others.



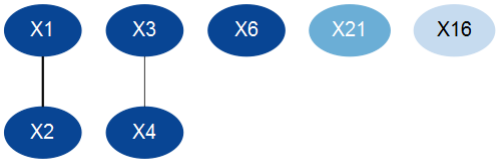
(a) A sampled graph for the original `corral` data set with 6 features



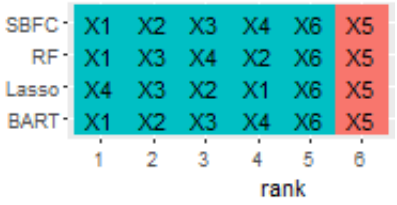
(b) A sampled graph for the augmented `corral` data set with 100 features



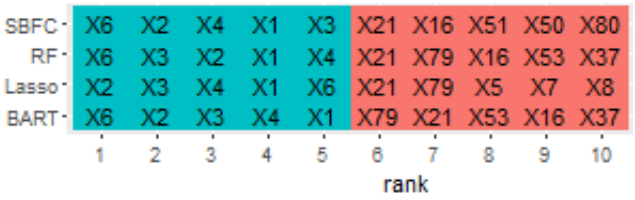
(c) Average graph for the original `corral` data set with 6 features



(d) Average graph for the augmented `corral` data set with 100 features

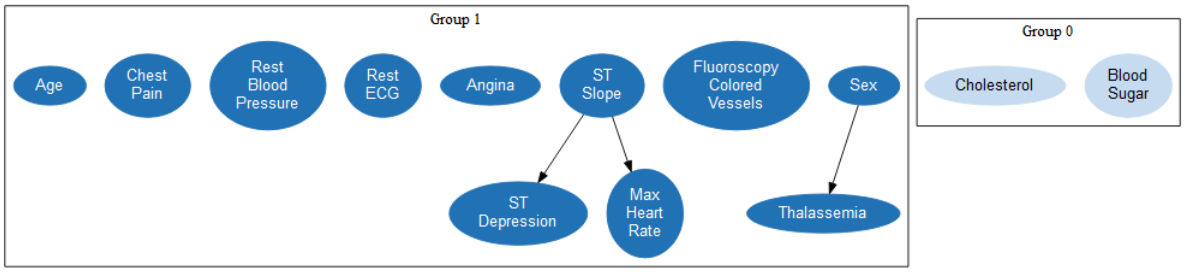


(e) Feature ranking comparison for the original `corral` data set with 6 features

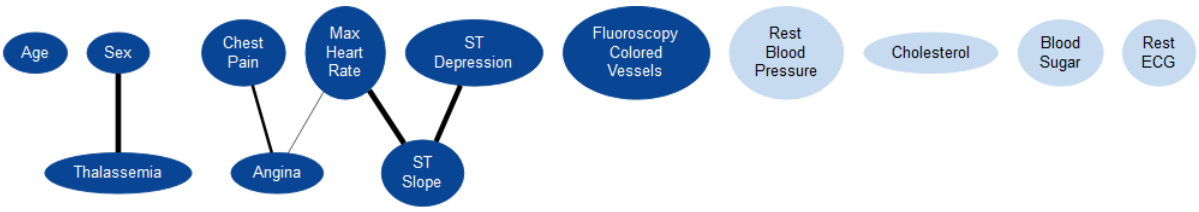


(f) Feature ranking comparison for the augmented `corral` data set with 100 features

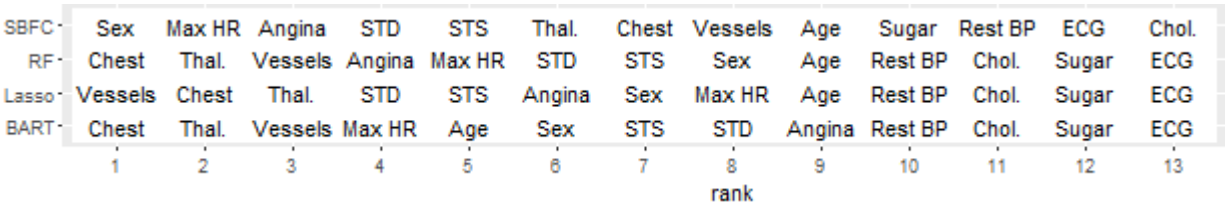
Figure 4: In the synthetic data set `corral`, the true feature structure is known: the relevant features are $\{X_1, X_2, X_3, X_4, X_6\}$, and the most relevant edges are $\{X_1, X_2\}$, $\{X_3, X_4\}$, while the other edges between the first 4 features are less relevant, and any edges with X_5 or X_6 are not relevant. The sampled graph in Figure 4a and the average graph in Figure 4c show that SBFC recovers the true correlation structure between the features, with the most relevant edges appearing the most frequently (as indicated by thickness). We generate extra noise features for this data set by choosing an existing feature at random and shuffling the rows, making it uncorrelated with the other features. The sampled graph in Figure 4b and the average graph in Figure 4d show that SBFC recovers the relevant features and some relevant interactions when the amount of noise increases. Figures 4e and 4f show that all the methods consistently rank the 5 relevant features (colored blue) above the rest (colored red).



(a) A sampled graph for `heart` data set

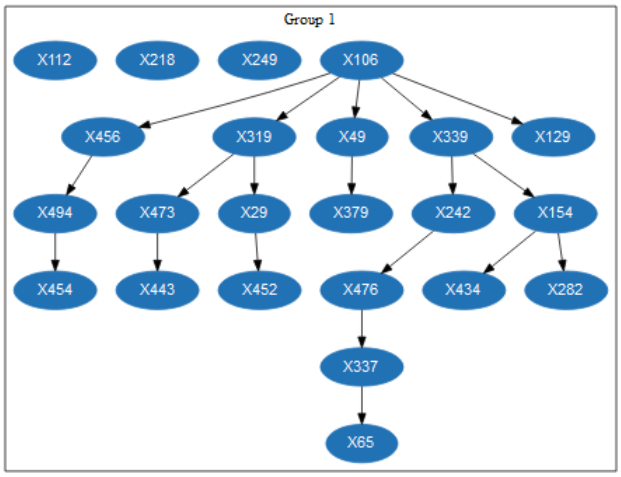


(b) Average graph for `heart` data set

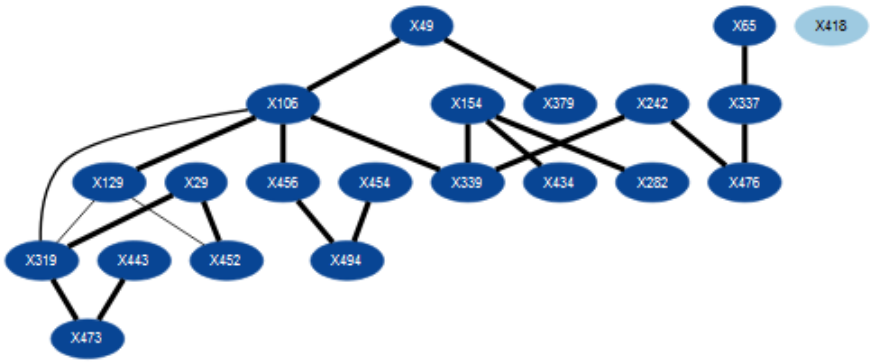


(c) Feature ranking comparison for `heart` data set

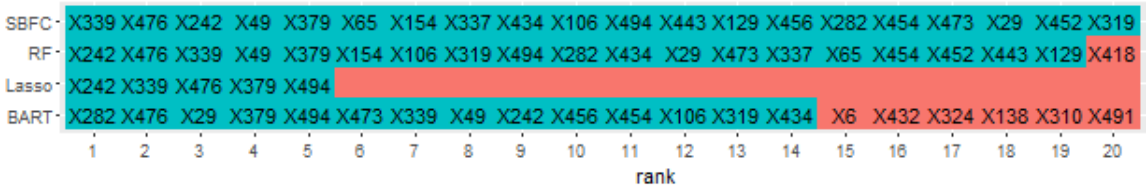
Figure 5: The sampled graph in Figure 5a and the average graph in Figure 5b show feature and interaction selection for the `heart` data set with features of medical significance. The dark-shaded features in the average graph are the most relevant for predicting heart disease. There are several groups of relevant interacting features: (Sex, Thalassemia), (Chest Pain, Angina), and (Max Heart Rate, ST Slope, ST Depression). The features in each group jointly affect the presence of heart disease. Figure 5c compares feature rankings with other methods, showing that all the methods agree on the top 9 features, but SBFC disagrees with the other methods on the top 3 features.



(a) A sampled graph for `madelon` data set

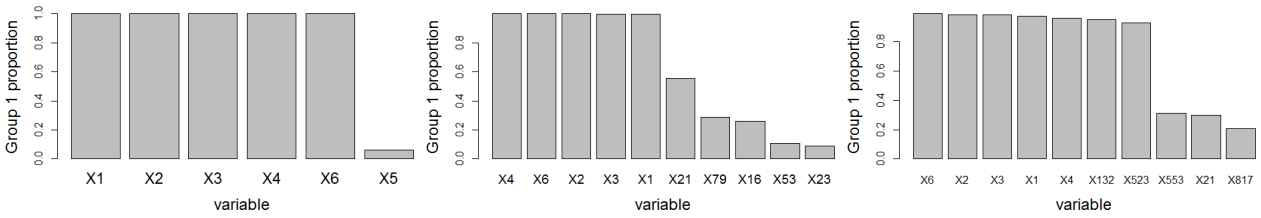


(b) Average graph for `madelon` data set

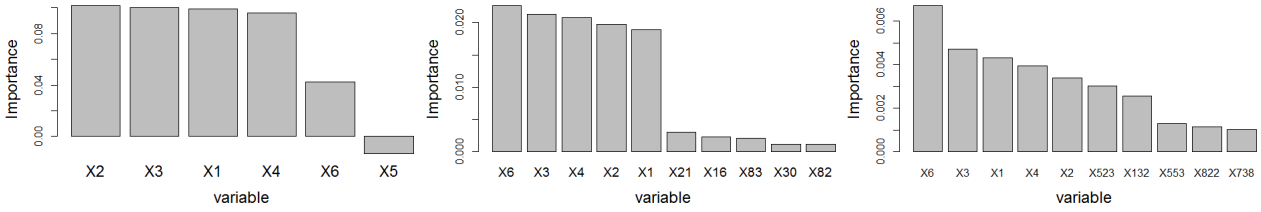


(c) Feature ranking comparison for `madelon` data set

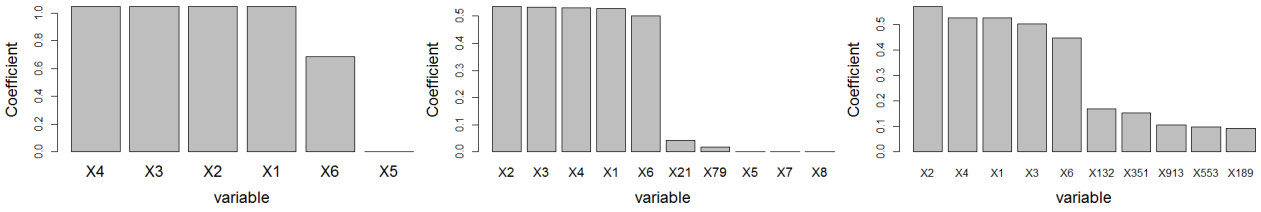
Figure 6: Feature and edge selection for the synthetic `madelon` data set, used in the 2003 NIPS feature selection challenge. This data set, with 20 relevant features and 480 noise features, was artificially constructed to illustrate the difficulty of selecting a feature set when no feature is informative by itself, and all the features are correlated with each other (Guyon et al., 2005). SBFC reliably selects the correct set of 20 relevant features (Guyon et al., 2006), as shown in Figure 6c, and appropriately puts them in a single connected component, shown in dark blue in the average graph in Figure 6b. As shown in Figure 6c, none of the other methods correctly identify the set of 20 relevant features (colored blue), though Random Forest comes close with 19 out of 20 correct. Our classification performance on this data set is not as good as that of BART or RF, likely because SBFC constrains these highly correlated features to form a tree, while a decision tree structure allows a feature to appear more than once.



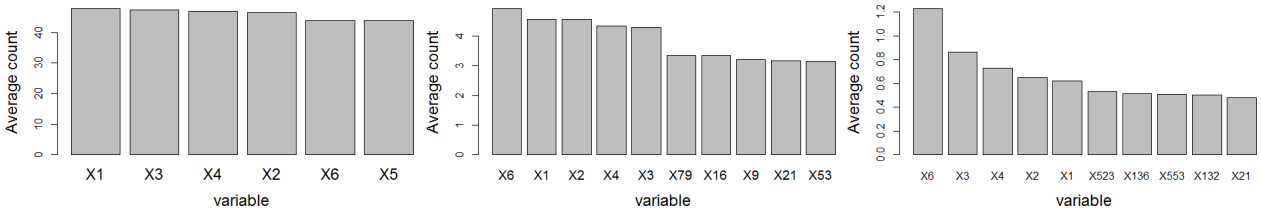
(a) SBFC for 6 features (b) SBFC for 100 features (c) SBFC for 1000 features



(d) RF for 6 features (e) RF for 100 features (f) RF for 1000 features

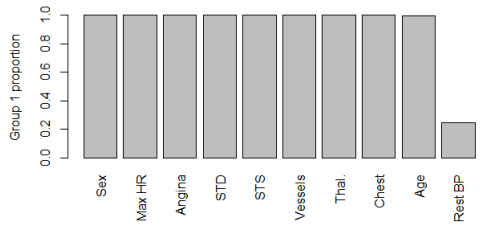


(g) Lasso for 6 features (h) Lasso for 100 features (i) Lasso for 1000 features

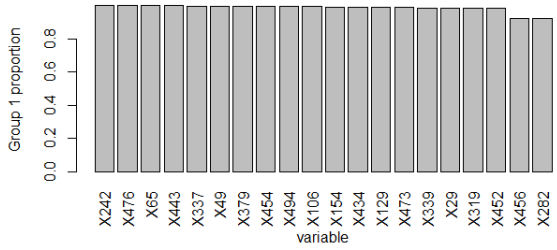


(j) BART for 6 features (k) BART for 100 features (l) BART for 1000 features

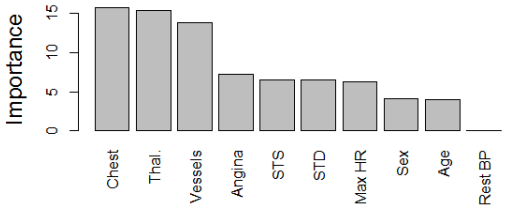
Figure 7: Feature selection comparison for the original and augmented `corral` data sets.



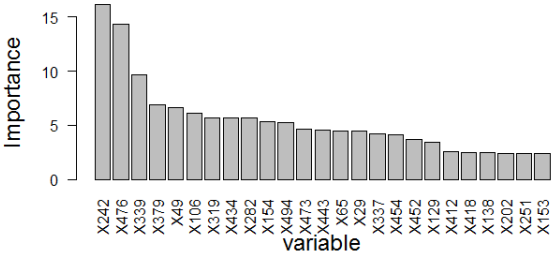
(a) SBFC on heart data set



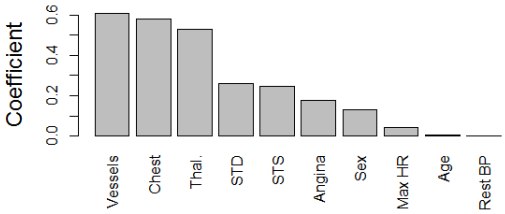
(b) SBFC on madelon data set



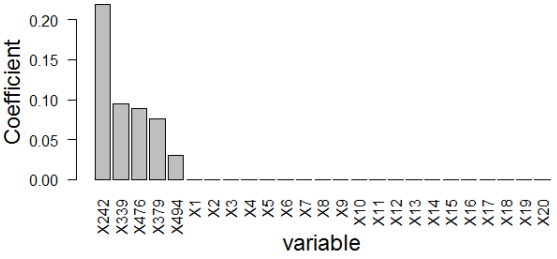
(c) RF on heart data set



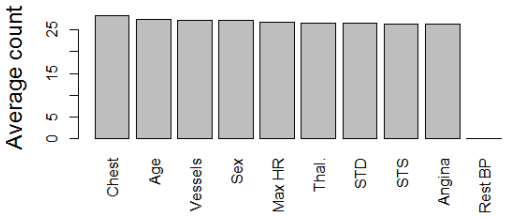
(d) RF on madelon data set



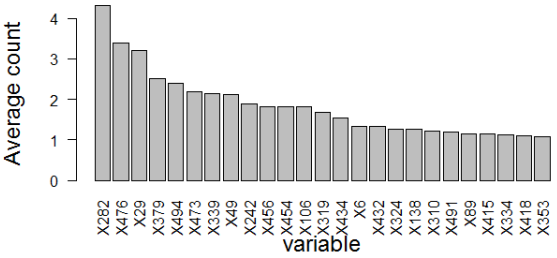
(e) Lasso on heart data set



(f) Lasso on madelon data set

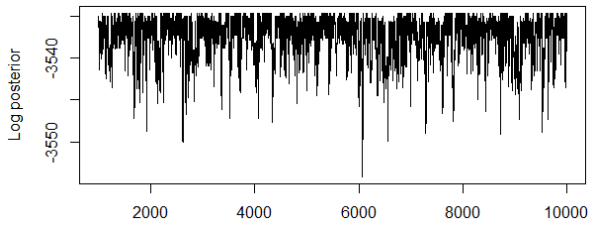


(g) BART on heart data set

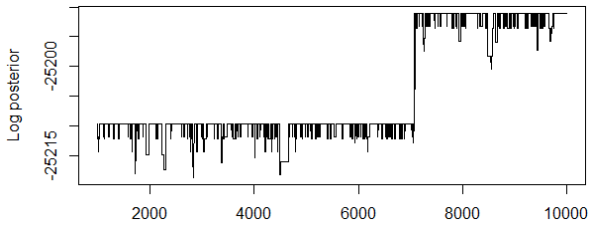


(h) BART on madelon data set

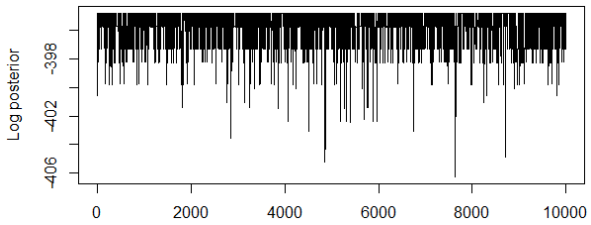
Figure 8: Feature selection comparison for the `heart` and `madelon` data sets.



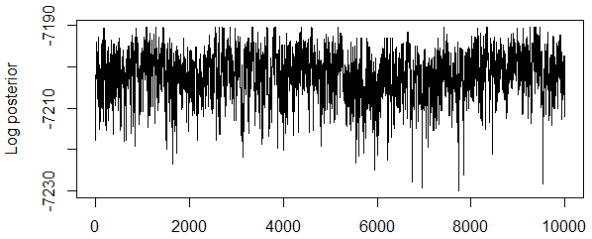
(a) `australian` data after 1000 iterations



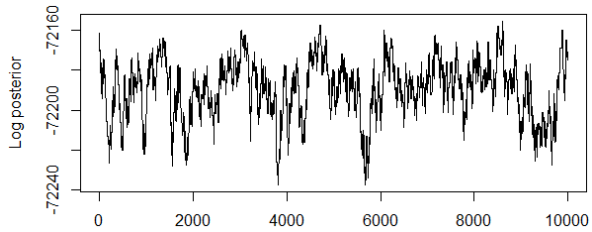
(b) `chess` data after 1000 iterations



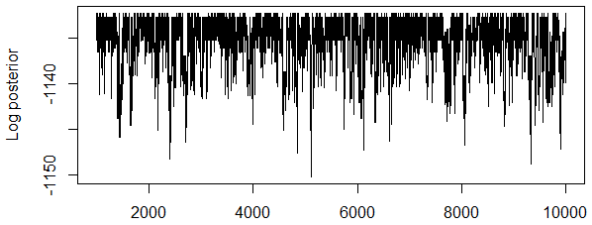
(c) original `corral` data



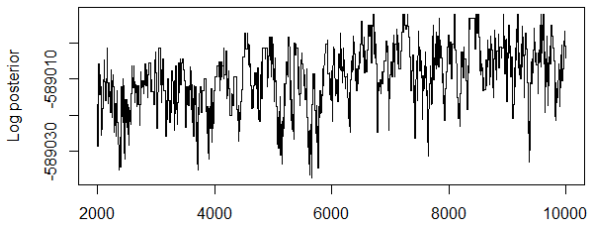
(d) augmented `corral` data with 100 features



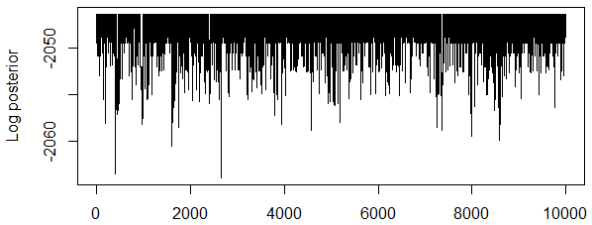
(e) augmented `corral` data with 1000 features



(f) `heart` data after 1000 iterations



(g) `madelon` data after 2000 iterations



(h) `mofn` data

Figure 9: Log posterior diagnostic plots for some data sets. Each run is 10000 iterations, with burn-in for the first 2000 iterations.

3.2.1 *Cluster dependence structure*

To test SBFC's capacity to pick up on relevant interactions in the data, we use a data set with several clusters of correlated features. The class label Y is binary, and the X_j take values in categories $1, 2, 3$. The signal group has two clusters of features, each associated with a particular class value, and the noise group has one cluster. The features are grouped into clusters as shown in Figure 10. There are 500 training cases and 1000 test cases.

The data generation process is as follows. First, we generate the features in the different clusters from independent multinomial distributions. In each signal group cluster, we synchronize the feature values with the associated class by sorting them in increasing or decreasing order for the instances with that class value. This makes the features correlated within the associated class, and independent outside the class, with their marginal distributions unchanged. In the noise group cluster, we randomly select half the instances, and sort the features over those instances. See Figure 5 for an example.

Table 6 shows that SBFC vastly outperforms the other methods in terms of classification accuracy on this simulation, and its accuracy does not decrease as more noise features are added to the data set, which happens to some of the other methods. As shown in Figure 11, SBFC reliably identifies the clusters and puts them in the correct groups, though as the number of noise features in the data set increases, it puts a few of those noise features into Group 1.

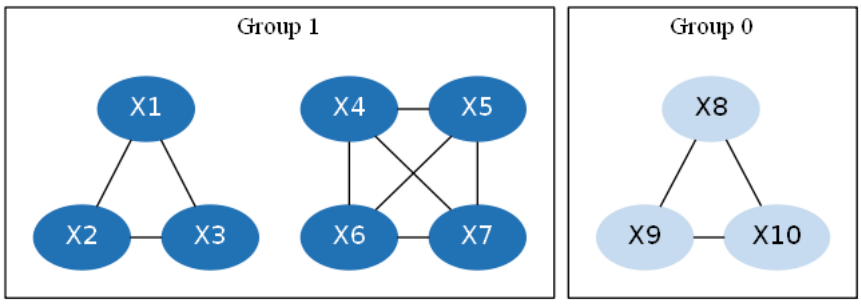


Figure 10: Cluster partition graph for the cluster simulation

Table 4: Cluster properties for the cluster simulation

Group	Cluster	features	Associated class
1	1	X_1, X_2, X_3	1
1	2	X_4, X_5, X_6, X_7	2
0	3	X_8, X_9, X_{10}	none
0	none	$X_{11}, \ldots, X_d$	none

Table 5: An example illustrating the cluster data generation process: we generate X_1 and X_2 independently, and then sort their values in class 1, in increasing and decreasing order respectively.

Before sorting:

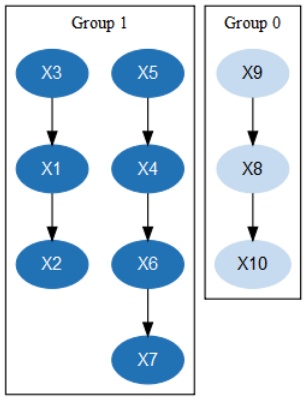
Class	**1**				**2**			
Instance	1	2	3	4	5	6	7	8
X_1	2	3	1	1	3	2	1	2
X_2	1	2	3	2	3	1	2	1

After sorting:

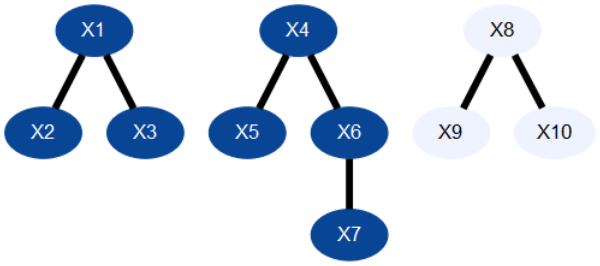
Class	**1**				**2**			
Instance	1	2	3	4	5	6	7	8
X_1	1	1	2	3	3	2	1	2
X_2	3	2	2	1	3	1	2	1

Table 6: Average classification accuracy on the cluster simulation

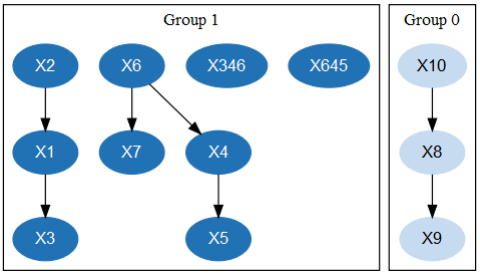
# features	BART	C5.0	CART	NB	LR	RF	SBFC	SVM
100	62.8	64.2	67.8	56	52.4	64.8	**93**	55.9
1000	51.3	51	49	40	42	52	**93**	51
10000	46	51	44	54	42	58	**92.3**	55



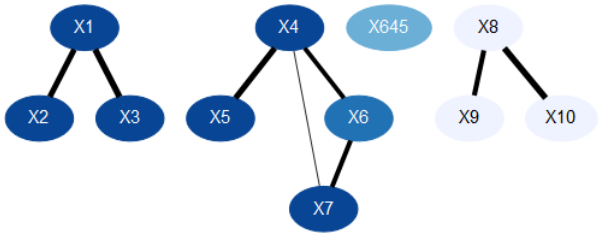
(a) A sampled graph for the cluster simulation with 100 features



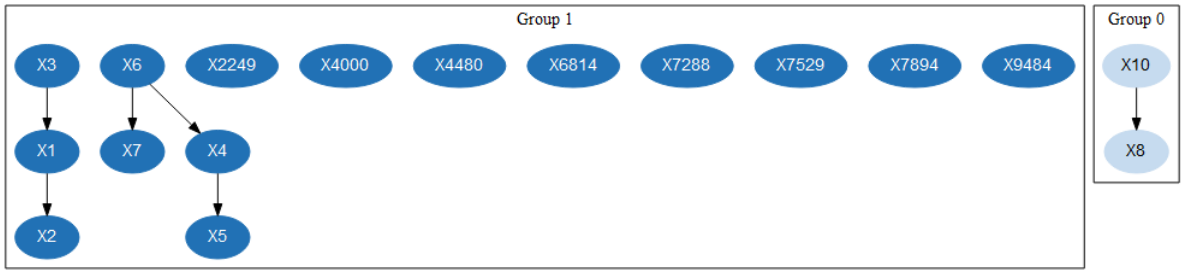
(b) Average graph for the cluster simulation with 100 features



(c) A sampled graph for the cluster simulation with 1000 features



(d) Average graph for the cluster simulation with 1000 features



(e) A sampled graph for the cluster simulation with 10000 features

(f) Average graph for the cluster simulation with 10000 features

Figure 11: SBFC graphs for the cluster simulation

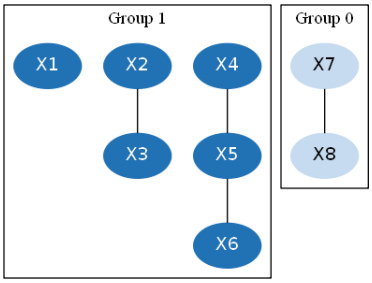


Figure 12: Logistic simulation structure

Table 7: Average classification accuracy on the logistic regression simulation

# features	BART	C5.0	CART	NB	LR	RF	SBFC	SVM
100	73.4	66.9	69.5	70.8	71.3	71.3	**76.1**	70.9
1000	73.7	67	63.3	63.7	53.2	65.8	**76.2**	65.6
10000	72.3	66.2	66	63	53.2	65.6	**75.1**	65.6

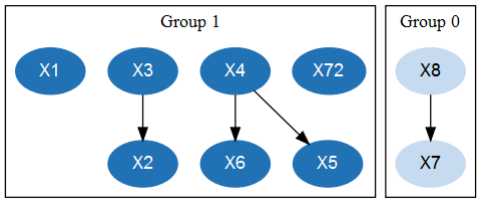
3.2.2 *Logistic regression model*

We use the following logistic regression model with binary features and class label:

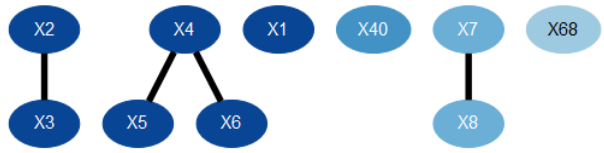
$$\text{logit}(P(Y = 1|\mathbf{X})) = X_1 - X_2 \cdot X_3 + X_4 \cdot X_5 - X_5 \cdot X_6$$

Like in the cluster simulation, we generate several sets of correlated features ($\{X_2, X_3\}$, $\{X_4, X_5, X_6\}$ and $\{X_7, X_8\}$), with features in different sets being independent, and generate Y based on the above model. The structure is shown in Figure 12. There are 500 training cases and 1000 test cases.

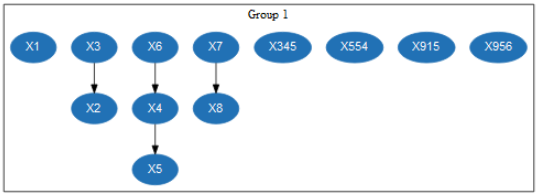
Table 7 shows that SBFC does better than the other methods except BART. Both SBFC and BART maintain a high accuracy as the number of noise features increases, while many of the other methods decrease in accuracy. As shown in Figure 13, SBFC correctly identifies relevant edges $X_2 - X_3$ and $X_4 - X_5$, but chooses $X_4 - X_6$ instead of $X_5 - X_6$. SBFC identifies the correlated features, but the noise edge $X_7 - X_8$ is



(a) A sampled graph for the logistic simulation with 100 features

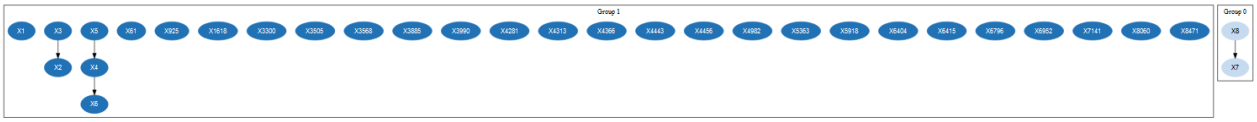


(b) Average graph for the logistic simulation with 100 features

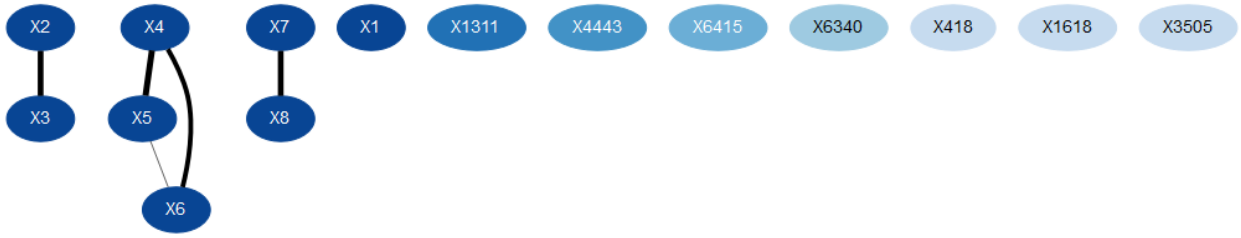


(c) A sampled graph for the logistic simulation with 1000 features

(d) Average graph for the logistic simulation with 1000 features

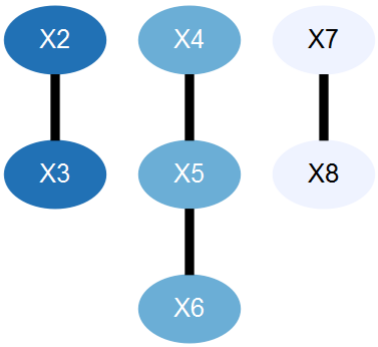


(e) A sampled graph for the logistic simulation with 10000 features

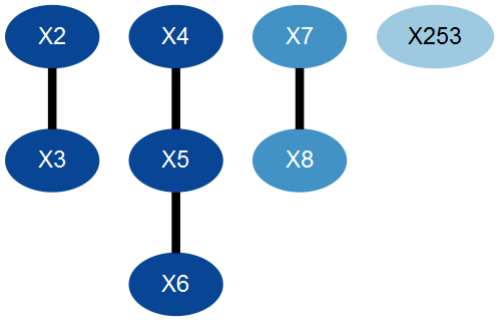


(f) Average graph for the logistic simulation with 10000 features

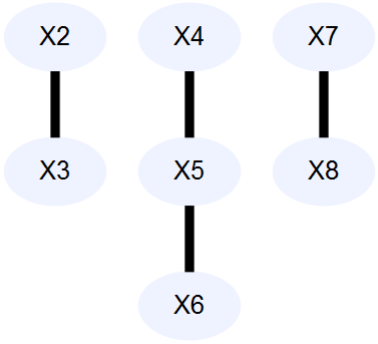
Figure 13: SBFC graphs for the logistic simulation.



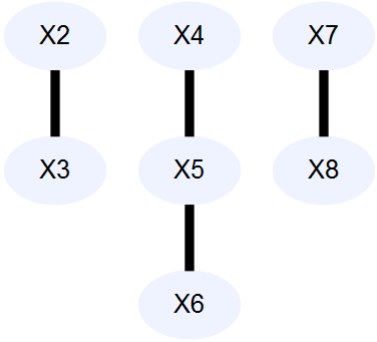
(a) Average graph for the noise simulation with 100 features, using default penalty $\lambda_y = 1$



(b) Average graph for the noise simulation with 1000 features, using default penalty $\lambda_y = 1$

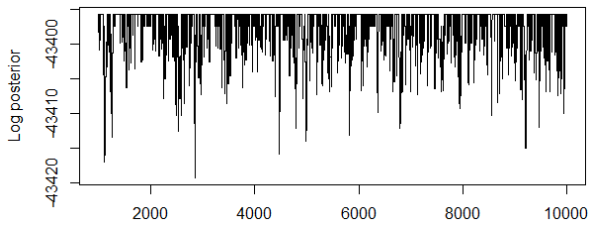


(c) Average graph for the noise simulation with 100 features, using $\lambda_y = 2$

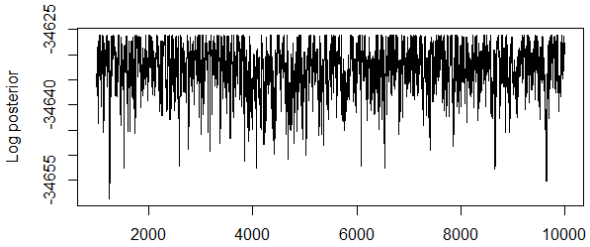


(d) Average graph for the noise simulation with 1000 features, using $\lambda_y = 2$

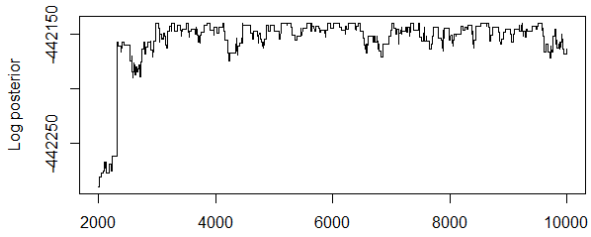
Figure 14: Average graphs on the noise simulation with different values of the y-edge penalty λ_y in the SBFC prior. The default penalty value of 1 leads to misclassifying the noise trees as signal, while setting the value to 2 avoids this issue.



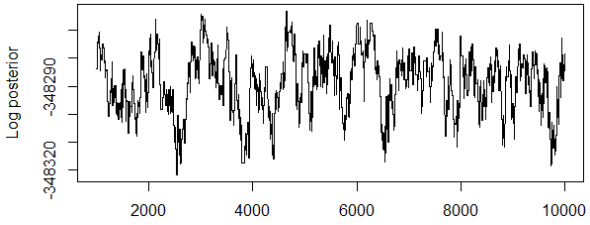
(a) Cluster simulation with 100 features after 1000 iterations



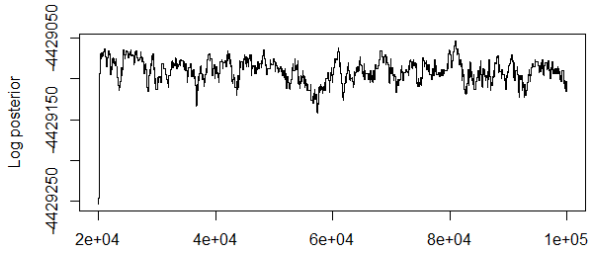
(b) Logistic simulation with 100 features after 1000 iterations



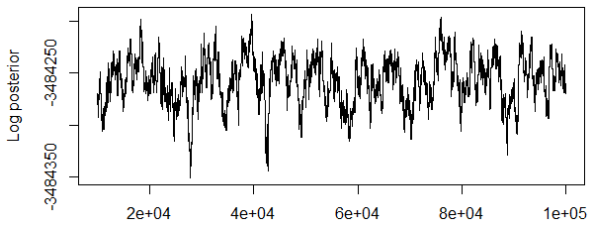
(c) Cluster simulation with 1000 features after 2000 iterations



(d) Logistic simulation with 1000 features after 1000 iterations



(e) Cluster simulation with 10000 features after 20000 iterations



(f) Logistic simulation with 10000 features after 10000 iterations

Figure 15: Log posterior diagnostic plots for the simulated data, with the cluster simulation in the first column and the logistic simulation in the second column. Each run is 10000 iterations for $d < 1000$ features and 100000 iterations for 10000 features, with burn-in for the first $1/5$ of the iterations.

increasingly misclassified as signal as we add more noise features. This suggests that the y-edge penalty λ_y in the prior is too small.

To investigate the issue with the misclassification of noise edges, we construct a version of this simulation where all the variables are noise, and the class label Y is simply distributed as $Bin(p = 0.5)$. As shown in Figure 14, all the noise trees get misclassified as signal for the default value of the y-edge penalty $\lambda_y = 1$, but increasing it to 2 leads to classifying the noise trees correctly. We should thus consider making this the default value of λ_y.

3.2.3 *MCMC diagnostics*

We examine MCMC diagnostics for the simulated data sets in Figure 15, which shows the log posterior trace plot for a single run on each data set. The default number of iterations is $\min(10000, 10d)$ where d is the number of features, with burn-in for $1/5$ of the iterations. Thus, for simulations with $d < 1000$ features, we run for 10000 iterations with 2000 iterations of burn-in, and for simulations with 10000 features, we run for 100000 iterations with 20000 iterations of burn-in. We observe that the mixing gets worse as the number of features increases from 100 to 1000, which suggests that it could make sense to scale the thinning factor with the number of features instead of using a constant value of 50 (though mixing does not continue to get worse from 1000 to 10000 features). For the cluster simulation, the default burn-in is not quite enough, and we see a jump right after burn-in both for 1000 and 10000 features.

3.3 R PACKAGE

The R package is available on CRAN and at `github.org/vkrakovna/sbfc`. The command for running the algorithm is `sbfc(data)`. It expects a discretized data set as input, which can be produced using the `data_disc()` command.

3.3.1 *Graph visualizations*

The `sbfc_graph()` command creates visualizations of the relevant features and feature interactions that were identified by the MCMC algorithm, such as those in Figures 4, 6, 12, etc. The command `sbfc_graph(average=FALSE, iter=N)` shows the sampled graph for the N-th MCMC iteration. The Group 1 nodes are dark-shaded, and the Group 0 nodes are light-shaded.

The command `sbfc_graph(average=TRUE)` shows an average graph, aggregating information from all MCMC samples. The nodes are color-coded according to relevance - the proportion of samples where the corresponding feature appeared in Group 1 (dark-shaded nodes appear more often). Edge thickness also corresponds to relevance - the proportion of samples where the corresponding feature interaction appeared. To avoid clutter, only edges that appear in at least a certain proportion of the sampled graphs are shown, specified by the `edge_cutoff` option, which defaults to 0.1. To see more of the low-relevance edges, lower the `edge_cutoff` value.

If you have a data set with meaningful variable labels, you can add these to your graph by setting the `labels` option to a set of labels of your choice. In the next section, we show how to do this for the `heart` data set.

How can we interpret these average graphs? To determine which groups of features influence the predictions the most, you can visually identify clusters of thicker edges in the graph. See Figure 5 for an example.

3.3.2 *Example: heart disease data*

The heart disease data set contains 13 variables related to heart disease, such as age, heart rate and blood pressure, and 270 observations. We are interested in identifying which features and feature interactions contribute to the presence of heart disease.

We begin by removing rows with missing values and discretizing the data set using Minimum Description Length partitioning. We then run SBFC on the discretized data set. Since the data set is a bit too small to divide into a training and test set, we leave the `n_train` argument for the number of training rows unspecified.

```
heart = data_disc(heart_data, n_train=NULL)
heart_result = sbfc(heart)
```

By default `sbfc()` runs 5-fold cross-validation if a test set is not supplied, and `sbfc_graph()` uses the MCMC samples from the first cross-validation fold. If you are not interested in classification and just want to get MCMC samples and graphs, you can run `sbfc(heart, cv=FALSE)`. Since this data set has variables with meaningful names, we supply these as node labels for the graph.

```
heart_labels = c("Age", "Sex", "Chest Pain", "Rest Blood Pressure",
    "Cholesterol", "Blood Sugar", "Rest ECG", "Max Heart Rate", "Angina",
    "ST Depression", "ST Slope", "Fluoroscopy Colored Vessels",
    "Thalassemia")
sbfc_graph(heart_result, labels=heart_labels, width=700)
```

To see the feature ranking, run `signal_var_proportion(heart_result, nvars)`. The proportion of MCMC samples where a specific variable appears in the signal group (Group 1) is called its *signal proportion*. The command returns the top `nvars` features in decreasing order of signal proportion.

To perform MCMC diagnostics as shown in Figures 9 and 15, plot the log posterior using `logposterior_plot(heart_result)`.

3.4 CONCLUSION

Selective Bayesian Forest Classifier is an integrated tool for supervised classification, feature selection, interaction detection and visualization. It splits the features into signal and noise groups according to their relationship with the class label, and uses tree structures to model interactions among both signal and noise features. The forest dependence structure gives SBFC modeling flexibility and competitive classification performance, and it maintains good feature and interaction selection performance as the signal to noise ratio decreases.

4

MINISPN: A MINIMALISTIC APPROACH TO SUM-PRODUCT NETWORK STRUCTURE LEARNING FOR REAL APPLICATIONS

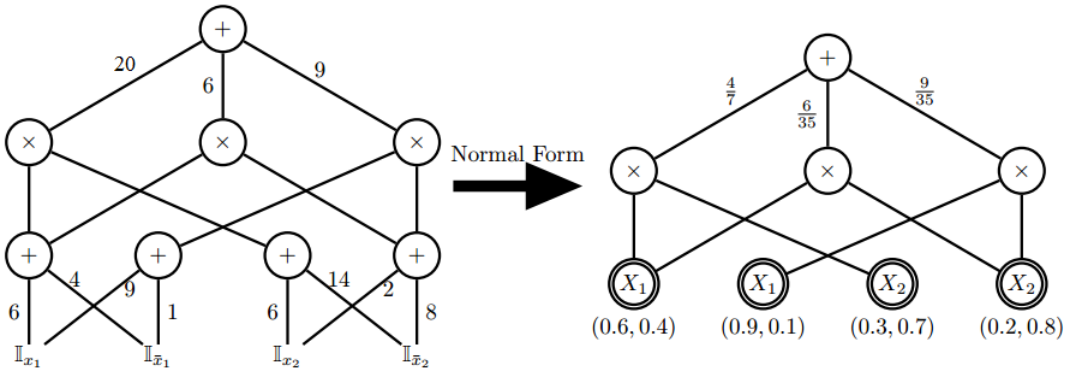


Figure 16: A sum-product network structure in indicator form and normal form[1]

4.1 INTRODUCTION

The Sum-Product Network (SPN) (Poon and Domingos, 2011) is a tractable and interpretable deep model for unsupervised learning. An advantage of SPNs over graphical models such as Bayesian networks is that they allow efficient exact inference in linear time with network size. An SPN represents a multivariate probability distribution with a directed acyclic graph consisting of sum nodes (weighted mixtures over instances), product nodes (partitions over features), and leaf nodes. An SPN can be represented either with feature indicators at the leaves (as in Poon and Domingos (2011)) or with univariate distributions over features at the leaves (as in Gens and Domingos (2013)),

[1] Figure from Zhao et al. (2015).

shown in Figure 16. We will be using the latter representation, also known as the normal form.

A key inference challenge in graphical models is summing an exponential number of products to compute the normalizing constant, also called the partition function. SPNs make this computation more efficient by reusing components via dynamic programming, thus removing the need for approximate inference (Poon and Domingos, 2011). The partition function Z is computed recursively from the leaves to the root: the value of a product node is the product of the values of its children ($Z_i = \prod_j Z_{ij}$), and the value of a sum node is the weighted sum of the values of its children ($Z_i = \sum_j w_{ij} Z_{ij}$). To compute the unnormalized probability of evidence in an SPN, we would perform the same computation with the univariate leaf distributions corresponding to known feature values replaced with delta functions at those values. When an SPN is converted to normal form, with the leaf distributions normalized and the sum node weights summing to 1, the normalizing constant becomes $Z = 1$ (Gens and Domingos, 2013).

An SPN is thus more tractable than a Bayesian network and remains no less interpretable while enjoying the expressive power of a deep architecture. It is assembled from interpretable components - products of independent groups of variables, mixtures of instances, and univariate distributions. Unlike in deep neural networks, the SPN architecture clearly shows how specific features influence intermediate components and the final prediction.

The standard algorithms for learning SPN structure assume discrete data with no missingness, and mostly test on the same set of benchmark data sets that satisfy these criteria. This is not a reasonable assumption when dealing with messy data sets in real applications. The Google Knowledge Graph (KG) is a semantic network of facts about the world, used to generate Knowledge Panels in Google Search. This data is quite heterogeneous, and a lot of it is missing, since much more is known about some entities in the graph than others. High missingness rates can also worsen the impact of

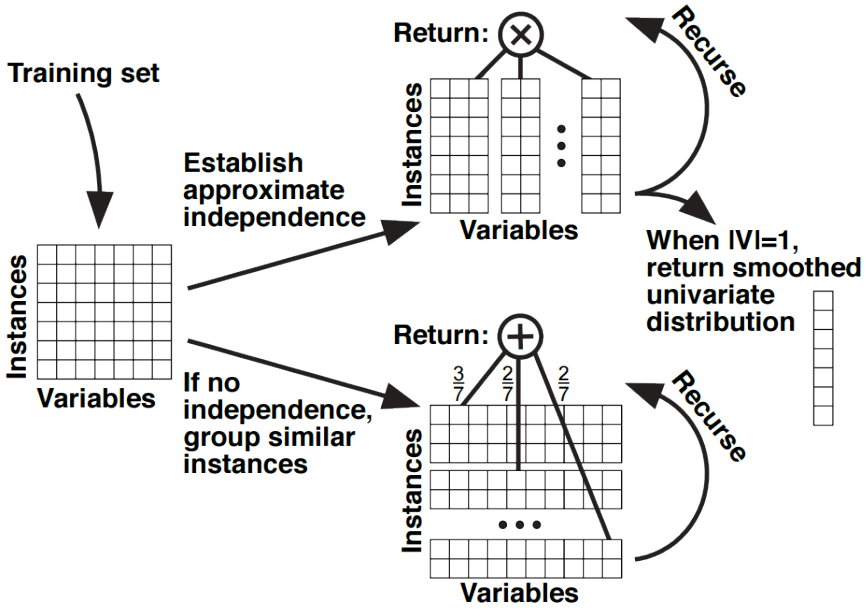


Figure 17: Recursive partitioning process in the LearnSPN algorithm[2]

discretizing continuous variables before doing structure learning, since that results in losing more of the already scarce covariance information.

Applications like the KG are common, and there is a need for an SPN learning algorithm that can handle this kind of data. In this paper, we present MiniSPN, a simplification of a state-of-the-art SPN learning algorithm (LearnSPN) that improves its applicability, performance and speed. We demonstrate the performance of MiniSPN on KG data and on standard benchmark data sets.

4.2 SPN LEARNING ALGORITHMS

4.2.1 *LearnSPN algorithm*

LearnSPN (Gens and Domingos, 2013) is a greedy algorithm that performs co-clustering by recursively partitioning variables into approximately independent sets and parti-

[2] Figure from Gens and Domingos (2013).

tioning the training data into clusters of similar instances. The variable and instance partitioning steps are applied to data slices (subsets of instances and variables) produced by previous steps.

The variable partition step uses pairwise independence tests on the variables, and the approximately independent sets are the connected components in the resulting dependency graph. The instance clustering step uses a naive Bayes mixture model for the clusters, where the variables in each cluster are assumed independent. The clusters are learned using hard EM with restarts, avoiding overfitting using an exponential prior on the number of clusters.

The splitting process continues until the data slice has too few instances to test for independence, at which point all the variables in that slice are considered independent. The end result is a tree-structured SPN. See Figure 17 for an overview of the recursive partitioning process.

The algorithm assumes that all the variables are discrete and there is no missing data. Hyperparameter values for the cluster penalty and the independence test critical value are determined using grid search.

Recent improvements of LearnSPN include the ID-SPN algorithm (Rooshenas and Lowd, 2014). ID-SPN follows the same recursive splitting procedure as LearnSPN until the number of instances or variables goes below a certain threshold. The remaining data slice becomes a multivariate leaf with an arithmetic circuit structure that is learned using a separate algorithm, aiming to capture indirect variable interactions. ID-SPN outperforms LearnSPN on benchmark data sets, but is more difficult to train.

4.2.2 *MiniSPN: a variation on LearnSPN*

The standard LearnSPN algorithm assumes that all the variables are discrete and there is no missing data. Hyperparameter values for the cluster penalty and the indepen-

dence test critical value are determined using grid search. The clustering step seems unnecessarily complex, involving a penalty prior, EM restarts, and hyperparameter tuning. It is by far the most complicated part of the algorithm in a way that seems difficult to justify, and likely the most time-consuming due to the restarts and hyperparameter tuning. We propose a variation on LearnSPN called MiniSPN that handles missing data, performs lazy discretization of continuous data in variable partition step, simplifies the model in the instance clustering step, and does not require hyperparameter search.

We simplify the naive Bayes mixture model in the instance clustering step by attempting a split into two clusters at any given point, and eliminating the cluster penalty prior, which results in a more greedy approach than in LearnSPN that does not require restarts or hyperparameter tuning. This seems like a natural choice of simplification - an extension of the greedy approach used at the top level of the LearnSPN algorithm. We compare a partition into univariate leaves to a mixture of two partitions into univariate leaves (generated using hard EM), and the split succeeds if the two-cluster version has higher validation set likelihood. If the split succeeds, we apply it to each of the two resulting data slices, and only move on to a variable partition step after the clustering step fails. The greedy approach is similar to the one used in the SPN-B method (Vergari et al., 2015), which however alternates between variable and instance splits by default, thus building even deeper SPNs.

In the variable partition step, we use the two-way Chi-square test of independence for each pair of variables. We perform the Chi-square test using the subset of rows where both variables are not missing, and the two variables are considered independent if the number of such rows is below threshold. The default threshold value of 20 is chosen to be consistent with the chi-square test validity cutoff, which requires the expected count in each of the 4 cells to be at least 5. The default critical value of 0.0005 is between the two critical values used in the standard algorithm (0.0001 and 0.0015). We apply binary binning to each continuous variable, using its median in the given

data slice as a cutoff (we tried using the mean as well, and got similar results). Note that the instance clustering step can handle continuous variables and does not require binning.

We allow multivariate leaves for continuous variables. When both the variable and instance splits fail, the set of variables is split into univariate Bernoulli and multivariate Gaussian leaf nodes.

4.2.3 *Pareto optimization algorithm*

This algorithm, previously used for learning SPN models on the Knowledge Graph, was inspired by the work of Grosse et al. (2012). It produces a Pareto-optimal set of models, trading off between degrees of freedom and validation set log likelihood score. At each iteration, production rules are randomly applied to add partition and mixture splits to the models in the current model set, and the new models are added to the model set. If a model in the model set has both lower degrees of freedom and higher log likelihood score than another model, the inferior model is removed from the set. The algorithm returns the model from the Pareto model set with the highest validation set log likelihood score.

4.3 EXPERIMENTS

4.3.1 *Experiments on Knowledge Graph data sets*

We use two data sets from the Knowledge Graph People collection. In the KG Professions data set, most of the variables are boolean indicators of whether each person belongs to a particular profession. There are 83 boolean variables and 4 continuous

Table 8: Knowledge Graph data set properties.

Data set	# Variables	Training set	Test set	Density	Missingness
Professions-10K	87	5000	5000	0.016	0.066
Professions-100K	87	50000	50000	0.015	0.066
Dates-10K	14	5000	5000	1	0.97
Dates-100K	14	50000	50000	1	0.94

Table 9: Average performance comparison on KG data sets using test set log likelihood.

Data set	Pareto	Hybrid	MiniSPN
Professions-10K	-10.2	-6.2	**-6.09**
Professions-100K	-6.61	-6.53	**-6.44**
Dates-10K	-8.66	**-8.53**	-8.68
Dates-100K	-17.1	-16.7	**-16.5**

Table 10: Average runtime comparison on KG data sets, in seconds.

Data set	Pareto	Hybrid	MiniSPN
Professions-10K	5.3	3.7	**0.4**
Professions-100K	72	131	**7.2**
Dates-10K	1.7	2.4	**0.26**
Dates-100K	29	566	**5.4**

variables. In the KG Dates data set, there are 14 continuous variables representing dates of life events for each person and their spouse(s), with most of the data missing. See Table 8 for details on data set properties. We use subsets of 10000 and 100000 instances from each of these data sets, and randomly split the data sets into a training and test set.

On the KG data sets, we compare MiniSPN, Pareto and Hybrid algorithms. We were not able to apply the standard LearnSPN algorithm on these data sets, since they contain missing data. Table 9 shows log likelihood performance on the test set, and Table 10 shows runtime performance (best performing methods are shown in bold). MiniSPN does better than Pareto, both in terms of log likelihood and runtime. Hybrid performs comparably to MiniSPN, but is usually the slowest of the three.

Table 11: Benchmark data set properties.[3]

Data set	Variables	Training set	Validation set	Test set	Density
NLTCS	16	16181	2157	3236	0.332
MSNBC	17	291326	38843	58265	0.166
KDDCup	65	180092	19907	34955	0.008
Plants	69	17412	2321	3482	0.18
Audio	100	15000	2000	3000	0.199
Jester	100	9000	1000	4116	0.608
Netflix	100	15000	2000	3000	0.541
Accidents	111	12758	1700	2551	0.291
Retail	135	22041	2938	4408	0.024
Pumsb-star	163	12262	1635	2452	0.27
DNA	180	1600	400	1186	0.253
Kosarak	190	33375	4450	6675	0.02
MSWeb	294	29441	3270	5000	0.01
Book	500	8700	1159	1739	0.016
EachMovie	500	4524	1002	591	0.059
WebKB	839	2803	558	838	0.064
Reuters-52	889	6532	1028	1540	0.036
20 Newsgroup	910	11293	3764	3764	0.049
BBC	1058	1670	225	330	0.078
Ad	1556	2461	327	491	0.008

4.3.2 *Experiments on benchmark data sets*

We use 20 benchmark data sets from the literature, exactly the same ones used in the original LearnSPN paper (Gens and Domingos, 2013). See Table 11 for a summary of data set properties. On these data sets, we are able to compare the performance of MiniSPN with the standard LearnSPN algorithm. We are particularly interested in the effect of MiniSPN's simple two-cluster instance split relative to the more complex instance split with the exponential prior and EM restarts used in the standard LearnSPN.

Table 12 shows log likelihood performance on the test set, and Table 13 shows runtime performance. Like on the KG data, we find that MiniSPN uniformly outperforms Pareto, and performs similarly to Hybrid and LearnSPN but runs much faster (on the

[3] Table from Gens and Domingos (2013).

Table 12: Performance on benchmark data sets in terms of test set log likelihood.

Data set	Pareto	Hybrid	MiniSPN	LearnSPN
NLTCS	-6.33	**-6.03**	-6.12	-6.1
MSNBC	-6.54	-6.4	-6.61	**-6.11**
KDDCup 2000	-2.17	**-2.13**	**-2.14**	-2.21
Plants	-17.3	**-13.1**	**-13.2**	**-13**
Audio	-41.9	**-39.9**	**-40**	-40.5
Jester	-54.6	**-52.9**	**-53**	-53.4
Netflix	-59.5	**-56.7**	**-56.8**	-57.3
Accidents	-40.4	-32.5	-32.6	**-30.3**
Retail	-11.1	**-11**	-11.1	-11.09
Pumsb-star	-40.8	-28.4	-28.3	**-25**
DNA	-98.1	-91.5	-93.9	**-89**
Kosarek	-11.2	**-10.8**	**-10.9**	-11
MSWeb	-10.7	**-9.94**	-10.1	-10.26
Book	-35.1	**-34.7**	**-34.7**	-36.4
EachMovie	-55	**-52.3**	**-52.2**	-52.5
WebKB	-161	**-155**	**-155**	-162
Reuters-52	-92	-85.2	**-84.7**	-86.5
20 Newsgroup	-156	**-152**	**-152**	-160.5
BBC	-258	-250	**-249**	-250
Ad	-52.3	-49.5	-49.2	**-22**

Table 13: Runtime comparison on benchmark data sets, in seconds.

Data set	Pareto	Hybrid	MiniSPN	LearnSPN
NLTCS	**4.8**	35	**1.4**	60
MSNBC	61	212	**5.6**	2400
KDDCup 2000	152	2080	**23**	400
Plants	**28**	780	**11**	160
Audio	**28**	556	**12**	955
Jester	**13**	193	**6.7**	1190
Netflix	**27**	766	**14**	1230
Accidents	**31**	1140	**18**	330
Retail	**25**	63	**7.3**	100
Pumsb-star	**47**	1100	**22**	350
DNA	**6.3**	45	**3**	300
Kosarek	90	537	**22**	200
MSWeb	75	572	**34**	260
Book	83	181	**32**	350
EachMovie	62	218	**22**	220
WebKB	**37**	169	**38**	900
Reuters-52	**76**	656	**95**	2900
20 Newsgroup	181	1190	**139**	28000
BBC	**33**	123	**42**	900
Ad	**58**	92	**50**	300

most time-intensive data set, 20 Newsgroup, MiniSPN takes 2 minutes while LearnSPN takes 8 hours). On the Ad data set, all our algorithms do much worse than the original LearnSPN. This might be related to the relatively small number of instances in this data set.

4.4 CONCLUSION

Sum-product networks have been receiving a lot of attention from researchers due to their expressiveness, efficient inference and interpretability.While other recent developments have mostly focused on improving performance on benchmark data sets, our variation on the classical LearnSPN learning algorithm is simple yet has a large impact on usability, by improving speed and making it possible to apply to messy real data sets in real applications.

5

INCREASING THE INTERPRETABILITY OF RECURRENT NEURAL NETWORKS USING HIDDEN MARKOV MODELS

5.1 INTRODUCTION

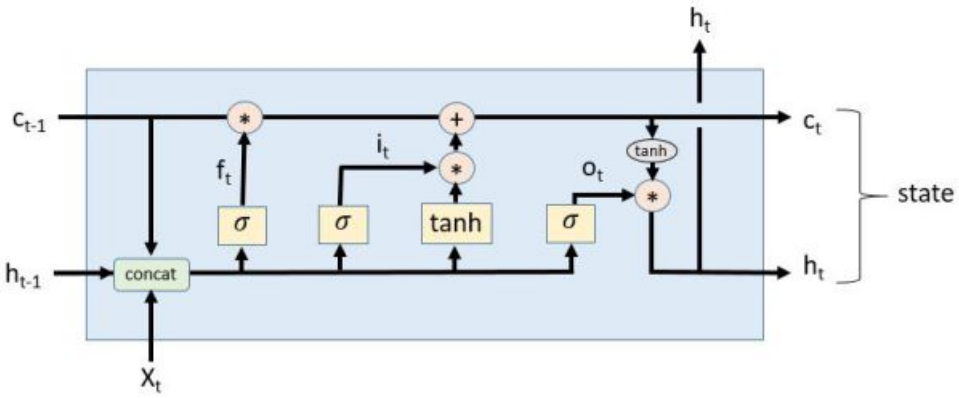
Following the recent progress in deep learning, researchers and practitioners of machine learning are recognizing the importance of understanding and interpreting what goes on inside these black box models. Recurrent neural networks have recently revolutionized speech recognition and translation, and these powerful models would be very useful in other applications involving sequential data. However, adoption has been slow in applications such as health care, where practitioners are reluctant to let an opaque system make crucial decisions. If we can make the inner workings of RNNs more interpretable, more applications can benefit from their power.

It is common for neural networks to show human-level performance most of the time, but also perform very poorly on seemingly easy cases. For instance, convolutional networks can misclassify adversarial examples with very high confidence (Nguyen et al., 2015), and made headlines in 2015 when the image tagging algorithm in Google Photos mislabeled African Americans as gorillas. We might expect recurrent networks to fail in similar ways as well. It would thus be useful to have more visibility into where these sorts of errors come from, e.g. which groups of features contribute to such flawed predictions.

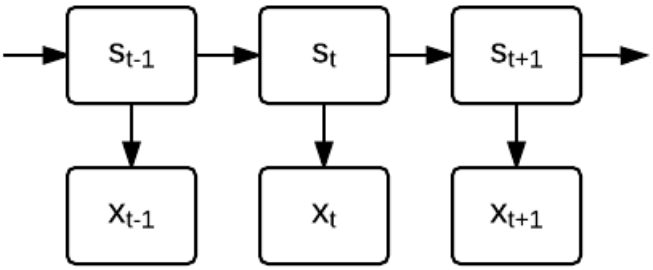
Several promising approaches to interpreting RNNs have been developed recently, focusing on a state-of-the-art RNN architecture called Long Short-Term Memory

(LSTM). Che et al. (2015) use gradient boosting trees to predict LSTM output probabilities and explain which features played a part in the prediction. They do not model the internal structure of the LSTM, but instead approximate the entire architecture as a black box. Karpathy et al. (2016) showed that in LSTM language models, around 10% of the memory state dimensions can be interpreted with the naked eye by color-coding the text data with the state values; some of them track quotes, brackets and other clearly identifiable aspects of the text. Building on these results, we take a somewhat more systematic approach to looking for interpretable hidden states, by using decision trees to predict individual hidden state dimensions (Figure 24). We visualize the overall dynamics of the hidden states by coloring the training data with the k-means clusters on the state vectors (Figures 21b, 22b, 23b).

We explore several methods for building interpretable models by combining LSTMs and HMMs. The existing body of literature mostly focuses on methods that specifically train the RNN to predict HMM states (Bourlard and Morgan, 1994) or posteriors (Maas et al., 2012), referred to as hybrid or tandem methods respectively. We first investigate an approach that does not require the RNN to be modified in order to make it understandable, as the interpretation happens after the fact. Here, we model the big picture of the state changes in the LSTM, by extracting the hidden states and approximating them with a continuous emission hidden Markov model (HMM). We then take the reverse approach where the HMM state probabilities are added to the output layer of the LSTM (see Figure 19). The LSTM model can then make use of the information from the HMM, and fill in the gaps when the HMM is not performing well, resulting in an LSTM with a smaller number of hidden states that could be interpreted individually (Figures 21, 22, 23).



(a) LSTM cell structure[1], where c_t, h_t and x_t represent the cell and hidden state vectors and the observations at time t respectively.

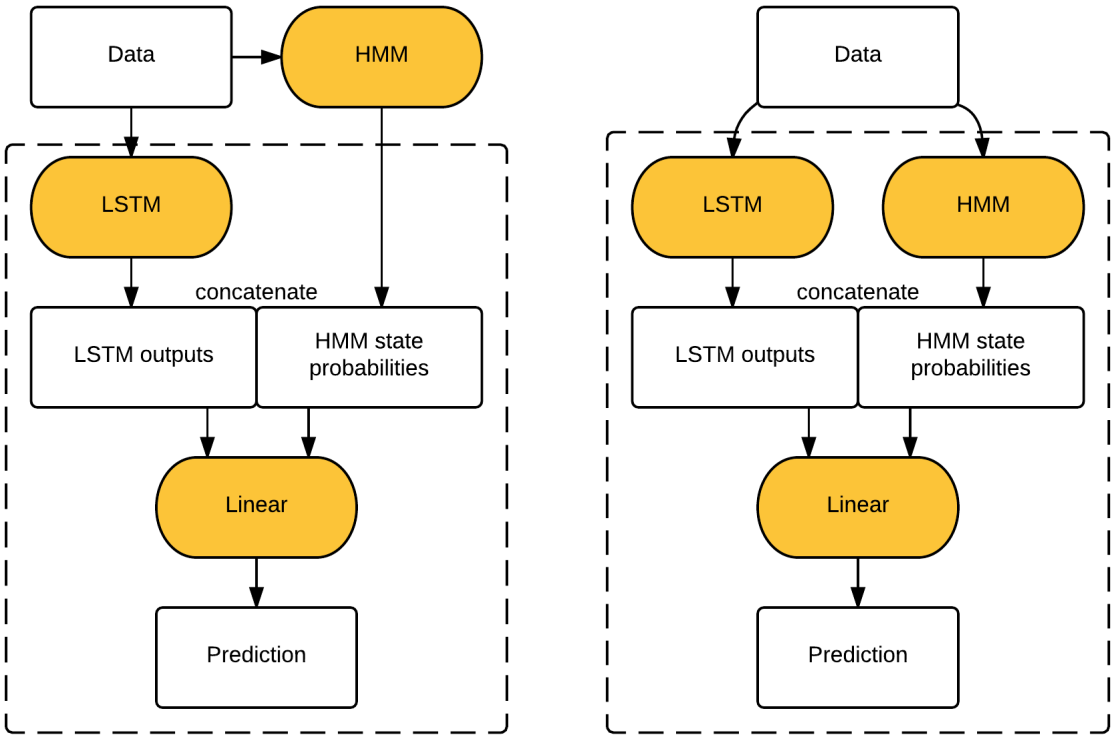


(b) Hidden Markov model, where s_t and x_t represent the hidden states and the observations at time t respectively.

5.2 METHODS

We compare hybrid HMM-LSTM models with an LSTM model, a continuous emission HMM (trained on the hidden states of a 2-layer LSTM), and a discrete emission HMM (trained directly on data).

[1] Figure by Chris Olah and Dennis Gannon.



(a) Sequentially trained hybrid algorithm (b) Jointly trained hybrid algorithm

Figure 19: Hybrid HMM-LSTM algorithms.

5.2.1 *Long Short-Term Memory models*

Long Short-Term Memory (LSTM) is an RNN variant designed to capture long-range interactions in the data. The LSTM cell structure is shown in Figure 18a. We use a character-level LSTM with 1 layer, based on the `rnn` Torch package from the Element-Research library. We train the LSTM for 10 epochs, starting with a learning rate of 1, and the learning rate is halved whenever $\exp(-l_t) > \exp(-l_{t-1}) + 1$, where l_t is the log likelihood score at epoch t. The L_2-norm of the parameter gradient vector is clipped at a threshold of 5.

5.2.2 *Hidden Markov models*

The Hidden Markov model is a classical model that is commonly considered transparent. It assumes that future observations and hidden states are independent from past ones given the current hidden state s_t. The HMM structure is shown in Figure 18b. Our HMM training procedure is as follows:

Initialization of HMM hidden states:

(Discrete HMM) Random multinomial draw for each time step (i.i.d. across time steps). (Continuous HMM) K-means clusters fit on LSTM state vectors, to speed up convergence relative to random initialization. We chose $k = 20$ based on a PCA analysis of the LSTM state vectors, as the first 20 components captured almost all the variance.

At each iteration:

1. Sample states using Forward Filtering Backwards Sampling algorithm (FFBS, Rao and Teh (2013)).

2. Sample transition parameters from a Multinomial-Dirichlet posterior. Let n_{ij} be the number of transitions from state i to state j. Then the posterior distribution of the i-th column of transition matrix T (corresponding to transitions from state i) is:

$$T_i \sim \text{Mult}(n_{ij}|T_i)\text{Dir}(T_i|\alpha)$$

   where α is the Dirichlet hyperparameter.

3. (Continuous HMM) Sample multivariate normal emission parameters from Normal-Inverse-Wishart posterior for state i:

$$\mu_i, \Sigma_i \sim N(y|\mu_i, \Sigma_i)N(\mu_i|0, \Sigma_i)\text{IW}(\Sigma_i)$$

(Discrete HMM) Sample the emission parameters from a Multinomial-Dirichlet posterior.

Evaluation:

We evaluate the methods based on how well they predict each observation in the validation set given preceding context. For the HMM models, we do a forward pass on the validation set (no backward pass unlike the full FFBS), and compute the HMM state distribution vector p_t for each time step t. Then we compute the predictive likelihood for the next observation as follows:

$$P(x_{t+1}|p_t) = \sum_{s_t=1}^{n} \sum_{s_{t+1}=1}^{n} p_{ts_t} \cdot T_{s_{t+1}s_t} \cdot P(x_{t+1}|s_{t+1})$$

where n is the number of hidden states in the HMM.

For the continuous emission HMM, we construct a discrete emission matrix based on the training outputs, which is used to compute the predictive likelihood.

5.2.3 *Hybrid models*

Our main hybrid model is put together sequentially, as shown in Figure 19a. We first run the discrete HMM on the data, outputting the hidden state distributions obtained by the HMM's forward pass, and then add this information to the architecture in parallel with a 1-layer LSTM. The linear layer between the LSTM and the prediction layer is augmented with an extra column for each HMM state. The LSTM component of this architecture can be smaller than a standalone LSTM in terms of the hidden state dimension, since it only needs to fill in the gaps in the HMM's predictions. The HMM is written in Python, and the rest of the architecture is in Torch.

We also build a joint hybrid model, as shown in Figure 19b, where the LSTM and HMM are simultaneously trained in Torch. We implemented an HMM Torch module, optimized using stochastic gradient descent rather than FFBS. Similarly to the sequential hybrid model, we concatenate the LSTM outputs with the HMM state probabilities.

The hybrid models have two types of hidden states: LSTM hidden states, indicated by h_t in Figure 18a, and HMM hidden states, indicated by s_t in Figure 18b. These are different quantities that are not to be confused with each other - the LSTM hidden state is a continuous multidimensional vector, while the HMM hidden state is a discrete categorical variable. When we refer to the number of LSTM state dimensions, we mean the dimension of the state vector, while when we refer to the number of HMM states, we mean the number of possible discrete states.

5.3 EXPERIMENTS

5.3.1 *Text data*

We test the models on several text data sets on the character level: the Penn Tree Bank (5M characters), and two data sets used by Karpathy et al. (2016), Tiny Shakespeare (1M characters) and Linux Kernel (5M characters).

Tables 14, 15 and 16 show the predictive loglikelihood of the next text character for each method. On all text data sets, the hybrid algorithm performs a bit better than the standalone LSTM with the same LSTM state dimension. This effect gets smaller as we increase the LSTM size and the HMM makes less difference to the prediction (though it can still make a difference in terms of interpretability). The hybrid algorithm with 20 HMM states does better than the one with 10 HMM states. The joint hybrid algorithm outperforms the sequential hybrid on Shakespeare data, but does worse on PTB and

Linux data, which suggests that the joint hybrid is more helpful for smaller data sets. The joint hybrid is an order of magnitude slower than the sequential hybrid, as the SGD-based HMM is slower to train than the FFBS-based HMM.

Unsurprisingly, the HMM baselines perform worse than the LSTM and hybrids, and, somewhat surprisingly, the discrete HMM outperforms the continuous HMM trained on LSTM state vectors. Figure 20 shows the convergence of the predictive likelihood for the HMM baselines. For the continuous HMM, the predictive likelihood actually goes down, so we expect that it's overfitting to the LSTM state vectors to the detriment of predicting the next character.

We interpret the HMM and LSTM states in the sequential hybrid algorithm with 10 LSTM state dimensions and 10 HMM states in Figures 21, 22 and 23, showing which features are identified by the HMM and LSTM components of the hybrid algorithm. In Figures 21a, 22a and 23a, we color-code the training data with the 10 HMM states. In Figures 21b, 22b and 23b, we apply k-means clustering to the 10-dimensional LSTM state vectors, and color-code the training data with the k-means clusters. The HMM and LSTM states pick up on spaces, indentation, and special characters in the data set (`<unk>` in Penn Tree Bank and comment symbols in Linux data). We see some examples where the HMM and LSTM complement each other, such as learning different things about spaces and comments on Linux data, or punctuation on the Shakespeare data.

We also use decision trees to interpret individual LSTM state dimensions in the sequential hybrid algorithm, fit using Classification and Regression Trees (R package `rpart`). While many of the LSTM state dimensions are distributed representations involving several kinds of features at once, such as vowels and numbers, we find some local representations focusing on specific features (see Figure 24). These tend to be the same features that come up in the k-means clustering over all the states, such as spaces in the PTB data and comment symbols in the Linux data. We hoped that the LSTM state dimensions of the hybrid algorithm would have lower complexity than the LSTM

Table 14: Predictive loglikelihood on Linux data (sorted by validation set performance).

Method	Parameters	LSTM state dimensions	HMM states	Validation	Training
Discrete HMM	1000		10	-2.76	-2.7
Discrete HMM	2000		20	-2.55	-2.5
LSTM	1215	5		-2.54	-2.48
Hybrid (joint)	2215	5	10	-2.35	-2.26
Hybrid	2215	5	10	-2.33	-2.26
Hybrid	3215	5	20	-2.25	-2.16
Hybrid (joint)	4830	10	10	-2.18	-2.08
LSTM	2830	10		-2.17	-2.07
Hybrid	3830	10	10	-2.14	-2.05
Hybrid	4830	10	20	-2.07	-1.97
LSTM	4845	15		-2.03	-1.9
Hybrid (joint)	5845	15	10	-2.00	-1.88
Hybrid	5845	15	10	-1.96	-1.84
Hybrid	6845	15	20	-1.96	-1.83
Hybrid (joint)	9260	20	10	-1.90	-1.76
LSTM	7260	20		-1.88	-1.73
Hybrid	8260	20	10	-1.87	-1.73
Hybrid	9260	20	20	-1.85	-1.71

Table 15: Predictive loglikelihood on Shakespeare data (sorted by validation set performance).

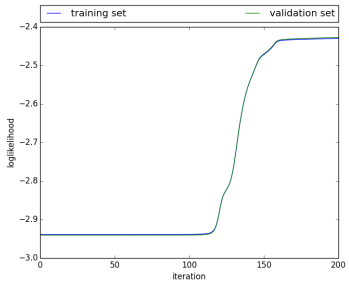
Method	Parameters	LSTM state dimensions	HMM states	Validation	Training
Continuous HMM	1300		20	-2.74	-2.75
Discrete HMM	650		10	-2.69	-2.68
Discrete HMM	1300		20	-2.5	-2.49
LSTM	865	5		-2.41	-2.35
Hybrid	1515	5	10	-2.3	-2.26
Hybrid	2165	5	20	-2.26	-2.18
LSTM	2130	10		-2.23	-2.12
Hybrid (joint)	1515	5	10	-2.21	-2.18
Hybrid	2780	10	10	-2.19	-2.08
Hybrid	3430	10	20	-2.16	-2.04
Hybrid	4445	15	10	-2.13	-1.95
Hybrid (joint)	3430	10	10	-2.12	-2.07
LSTM	3795	15		-2.1	-1.95
Hybrid	5095	15	20	-2.07	-1.92
Hybrid	6510	20	10	-2.05	-1.87
Hybrid (joint)	4445	15	10	-2.03	-1.97
LSTM	5860	20		-2.03	-1.83
Hybrid	7160	20	20	-2.02	-1.85
Hybrid (joint)	7160	20	10	-1.97	-1.88

Table 16: Predictive loglikelihood on Penn Tree Bank data (sorted by validation set performance).

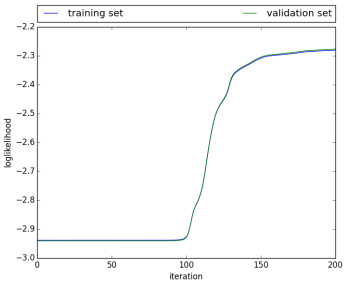
Method	Parameters	LSTM state dimensions	HMM states	Validation	Training
Continuous HMM	1000	100	20	-2.58	-2.58
Discrete HMM	500		10	-2.43	-2.43
Discrete HMM	1000		20	-2.28	-2.28
LSTM	715	5		-2.22	-2.22
Hybrid	1215	5	10	-2.14	-2.15
Hybrid (joint)	1215	5	10	-2.08	-2.08
Hybrid	1715	5	20	-2.06	-2.07
LSTM	1830	10		-1.99	-1.99
Hybrid	2330	10	10	-1.94	-1.95
Hybrid (joint)	2830	10	10	-1.94	-1.95
Hybrid	2830	10	20	-1.93	-1.94
LSTM	3345	15		-1.82	-1.83
Hybrid	3845	15	10	-1.81	-1.82
Hybrid	4345	15	20	-1.8	-1.81
Hybrid (joint)	6260	20	10	-1.73	-1.74
LSTM	5260	20		-1.72	-1.73
Hybrid	5760	20	10	-1.72	-1.72
Hybrid	6260	20	20	-1.71	-1.71

Table 17: Decision tree size and depth comparison on the text data sets, showing mean and standard deviation over all LSTM state dimensions (sorted by number of parameters).

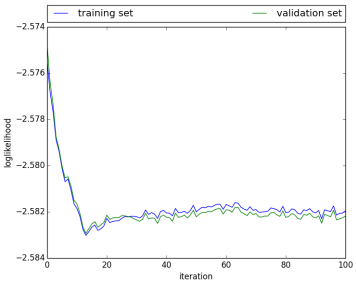
Data	Method	Parameters	LSTM state dimensions	HMM states	Tree depth	Tree size
Penn Tree Bank	LSTM	1830	10		5.1 ± 0.94	20.6 ± 5.92
	Hybrid	2330	10	10	4.6 ± 0.66	20.8 ± 5.01
	Hybrid	2830	10	20	4.3 ± 0.46	20.4 ± 5.06
	LSTM	5260	20		4.85 ± 1.06	20.3 ± 4.62
	Hybrid	5760	20	10	5 ± 1	21 ± 5.18
	Hybrid	6260	20	20	5.05 ± 0.86	21.1 ± 5.11
Shak.	LSTM	2130	10		4.5 ± 0.5	19.2 ± 2.75
	Hybrid	2780	10	10	4.1 ± 0.83	17.8 ± 4.12
	LSTM	5860	20		4.7 ± 0.84	19.7 ± 3.05
	Hybrid	6510	20	10	4.75 ± 0.94	21.1 ± 4.62
Linux Kernel	LSTM	2830	10		4.3 ± 0.64	17.2 ± 3.6
	Hybrid	3830	10	10	4.8 ± 0.98	18.8 ± 3.74
	Hybrid	4830	10	20	4.7 ± 0.78	19.4 ± 4.7
	LSTM	7260	20		4.85 ± 0.85	19.2 ± 4.2
	Hybrid	8260	20	10	4.8 ± 0.92	19.7 ± 5.15
	Hybrid	9260	20	20	4.75 ± 0.77	21 ± 4.64



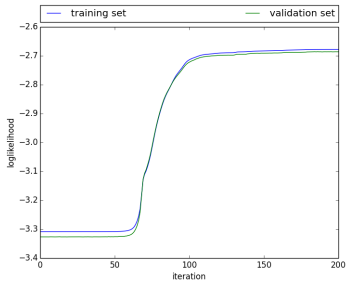
(a) Discrete HMM with 10 states on PTB



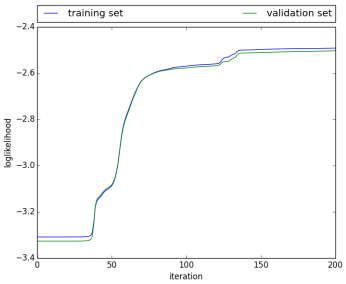
(b) Discrete HMM with 20 states on PTB



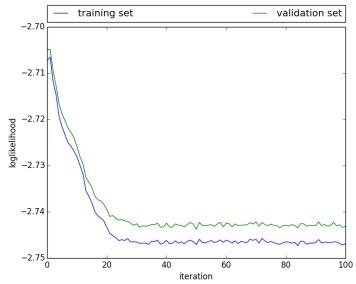
(c) Continuous HMM with 20 states on PTB



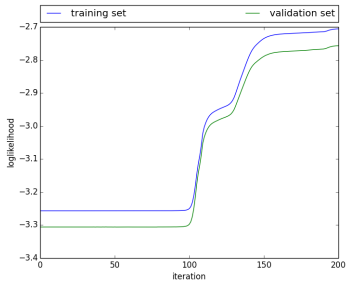
(d) Discrete HMM with 10 states on Shakespeare



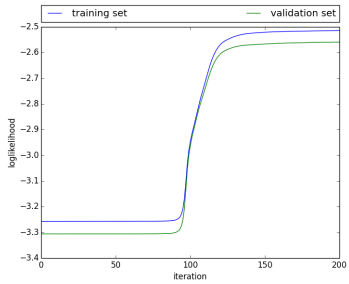
(e) Discrete HMM with 20 states on Shakespeare



(f) Continuous HMM with 20 states on Shakespeare



(g) Discrete HMM with 10 states on Linux



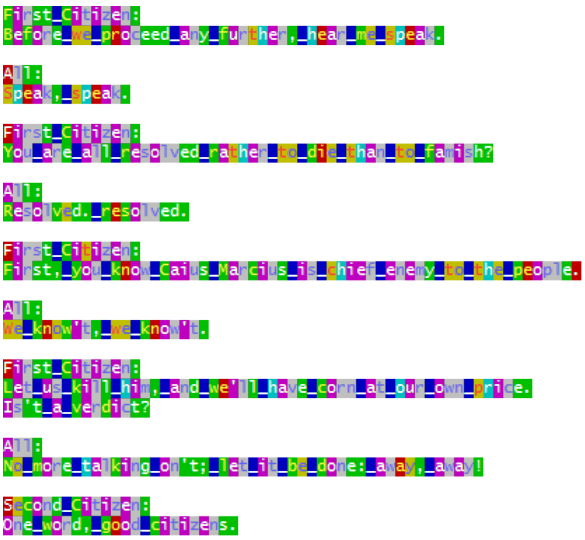
(h) Discrete HMM with 20 states on Linux

Figure 20: Convergence of the predictive likelihood for the discrete and continuous HMM baselines (training set in blue, validation set in green).

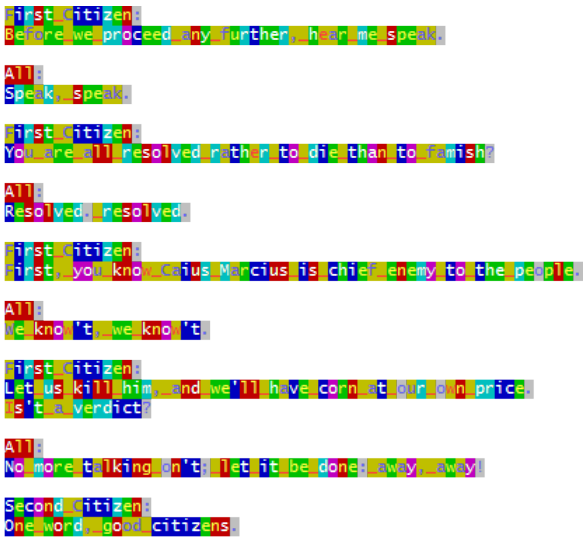
(a) Hybrid HMM component: colors correspond to 10 HMM states. The green cluster (with yellow font) identifies spaces, and the red cluster (with yellow font) identifies <unk>. The purple cluster picks up on some vowels and apostrophes.

(b) Hybrid LSTM component: colors correspond to 10 k-means clusters on hidden state vectors. The purple cluster picks up on spaces and some letters, and the green cluster (with white font) identifies <unk> and some letters.

Figure 21: Visualizing HMM and LSTM states on PTB data for the hybrid algorithm with 10 LSTM state dimensions and 10 HMM states. The HMM and LSTM components learn some complementary features in the text: while the HMM clearly distinguishes spaces and the unknown character <unk>, the LSTM does not need to learn these features very well because they were already learned by the HMM.

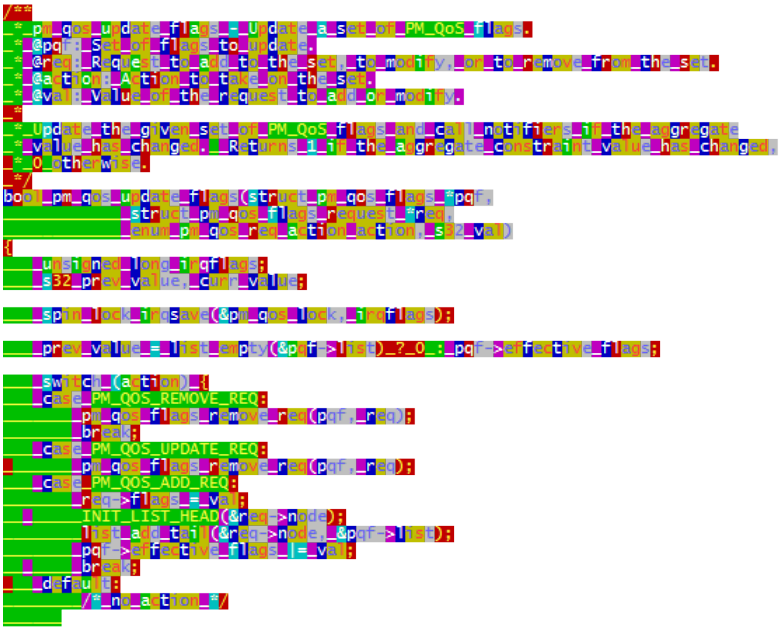


(a) Hybrid HMM component: colors correspond to 10 HMM states. Blue cluster identifies spaces. Green cluster (with white font) identifies punctuation and ends of words. Purple cluster picks up on some vowels.

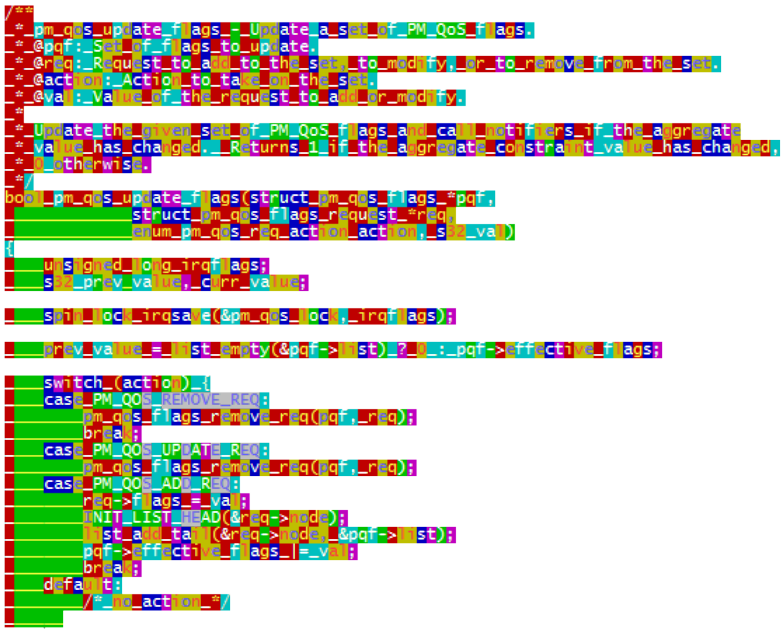


(b) Hybrid LSTM component: colors correspond to 10 k-means clusters on hidden state vectors. Yellow cluster (with red font) identifies spaces. Grey cluster identifies punctuation (except commas).

Figure 22: Visualizing HMM and LSTM states on Shakespeare data for the hybrid algorithm with 10 LSTM state dimensions and 10 HMM states. The HMM and LSTM components learn some complementary features in the text: while both learn to identify spaces, the LSTM does not completely identify punctuation or pick up on vowels, which the HMM has already done.

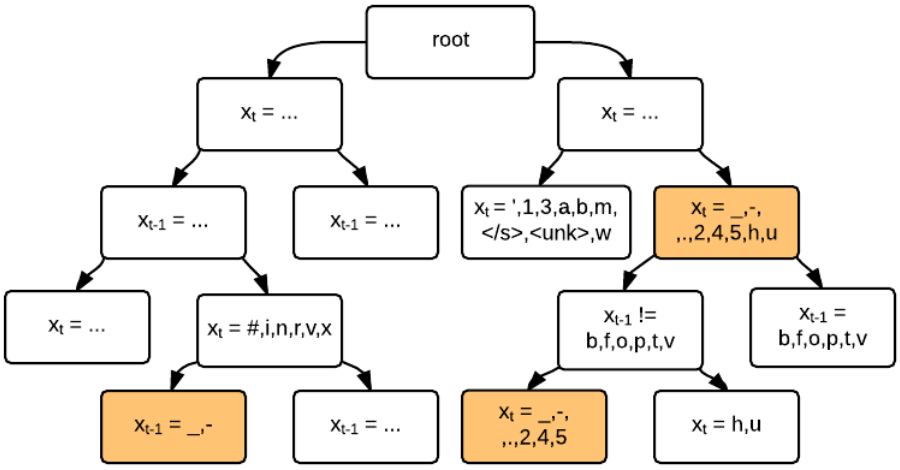


(a) Hybrid HMM component: colors correspond to 10 HMM states. Distinguishes comments and indentation spaces (green with yellow font) from other spaces (purple). Red cluster (with yellow font) identifies punctuation and brackets.

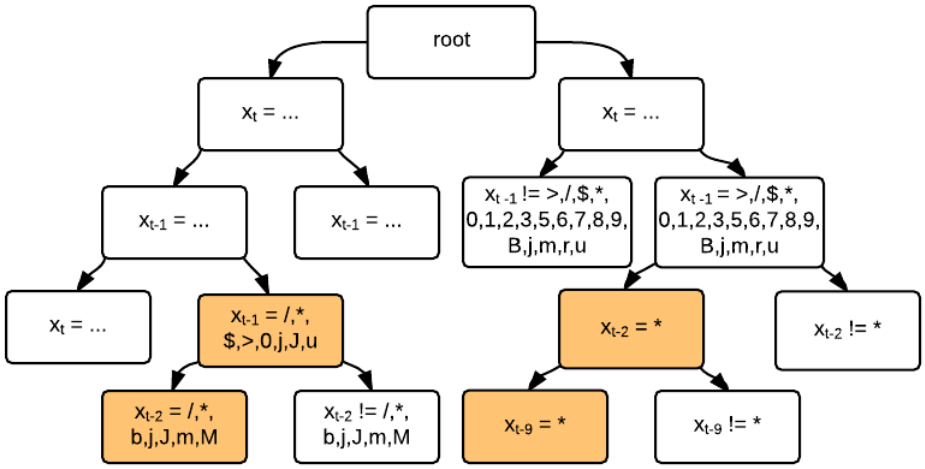


(b) Hybrid LSTM component: colors correspond to 10 k-means clusters on hidden state vectors. Distinguishes comments, spaces at beginnings of lines, and spaces between words (red with white font) from indentation spaces (green with yellow font). Opening brackets are red (yellow font) and closing brackets are green (white font).

Figure 23: Visualizing HMM and LSTM states on Linux data for the hybrid algorithm with 10 LSTM state dimensions and 10 HMM states. The HMM and LSTM components learn some complementary features in the text related to spaces and comments.



(a) Decision tree example on PTB data. The hidden states of the 10-state hybrid mostly track spaces and numbers.



(b) Decision tree example on Linux data. The hidden states of the 10-state hybrid mostly track comment characters.

Figure 24: Decision trees predicting individual hidden states of the hybrid algorithm based on the 10 preceding characters $x_t, x_{t-1}, \ldots, x_{t-9}$. Overly large and unintuitive splits were replaced with ....

state dimensions in the standalone LSTM, as measured by the size and depth of the approximating decision trees, but this turned out not to be the case. As shown in Table 17, the differences in tree depth and size between the LSTM and the hybrid are not significant, as the means are within two standard deviations of each other.

While the hybrid algorithm performs somewhat better than an LSTM with the same LSTM state dimension, it also has more parameters. The number of parameters for an LSTM with n_s states on a text data set with n_c characters is $n_s \cdot (8n_s + 3)$ for the LSTM component, plus $n_s \cdot n_c$ for each of the lookup table and linear layers. The hybrid algorithm with n_h HMM states has an additional $n_c \cdot n_h$ parameters in the linear layer. The HMM baseline usually has the fewest parameters but also much worse performance. We have found that a hybrid and an LSTM with the same number of parameters perform similarly, however you could argue that the extra parameters in the linear layer of the hybrid architecture are more interpretable than the parameters of various weight matrices inside the LSTM cell.

5.3.2 *Multivariate time series data*

While natural language processing is the most common application for RNNs, we are also interested in applying our interpretable hybrid models to multivariate time series data, such as medical time series. Here, the observations at each time step are multivariate and discrete - we make a simplifying assumption that they are binary and independent of each other. The response variable (output) is also binary and distinct from the observations, while in the text data the output was the observation at the next time step. We investigate our performance on synthetic data and ICU time series data.

Synthetic data

As a sanity check for the relative strengths and weaknesses of the methods, we generate several synthetic data sets - one designed to be easy for an HMM, one designed to be easy for an LSTM, and a combination of the two. In this setup, we have hidden states $\boldsymbol{s_t}$ (represented by indicator vectors), multivariate observations $\boldsymbol{x_t}$ and outputs o_t for each time step t. The hidden states and observations are related as follows:

$$\boldsymbol{s_t} = \text{Mult}(n = 1, \boldsymbol{p} = \tilde{T} \cdot \boldsymbol{s_{t-1}})$$

$$\boldsymbol{x_t} = \text{Bin}(n = 1, \boldsymbol{p} = \boldsymbol{s_t'}\tilde{E}),$$

where the observations in $\boldsymbol{x_t}$ are independent Binomial variables given the hidden state. We have 5 hidden states and 7 observations, with the following transition and emission matrices:

$$\tilde{T} = \begin{vmatrix} .5 & .25 & 0 & 0 & 0 \\ .5 & .5 & .25 & 0 & 0 \\ 0 & .25 & .5 & .25 & 0 \\ 0 & 0 & .25 & .5 & .5 \\ 0 & 0 & 0 & .25 & .5 \end{vmatrix}, \tilde{E} = \begin{vmatrix} .5 & .5 & .5 & 0 & 0 & 0 & 0 \\ 0 & .5 & .5 & .5 & 0 & 0 & 0 \\ 0 & 0 & .5 & .5 & .5 & 0 & 0 \\ 0 & 0 & 0 & .5 & .5 & .5 & 0 \\ 0 & 0 & 0 & 0 & .5 & .5 & .5 \end{vmatrix}$$

The outputs are generated using weight vectors $\boldsymbol{w_s}$ and $\boldsymbol{w_x}$ applied to the states and observations respectively, and a bias weight w_{bias} that affects the proportion of 0 and 1 output values:

$$o_t = \text{Bin}\left(n = 1, p = \frac{1}{1 + \exp(-\boldsymbol{w_s} \cdot \boldsymbol{s_t} - \boldsymbol{w_x} \cdot \boldsymbol{x_t} - w_{bias})}\right)$$

The default weight vector values are $\boldsymbol{w_s} = [0, 0, a, 0, 0]$ and $\boldsymbol{w_x} = [0, 0, b, 0, b, 0, 0]$, with a bias term $w_{bias} = 2$ that skews the o_t values towards 0. To create the data set

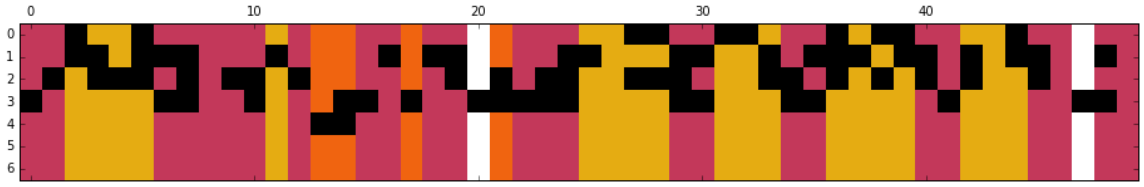
Table 18: Predictive loglikelihood on the synthetic multivariate binary data sets (sorted by validation set performance).

Data	Method	Parameters	LSTM state dimensions	HMM states	Validation	Training
D_s	LSTM	265	5		-0.51	-0.51
	Hybrid (joint)	275	10	5	-0.5	-0.5
	Hybrid	275	5	5	-0.48	-0.47
	HMM	60		5	-0.47	-0.47
D_{sx}	LSTM	265	5		-0.56	-0.56
	HMM	60		5	-0.56	-0.56
	Hybrid (joint)	275	10	5	-0.54	-0.54
	Hybrid	275	5	5	-0.53	-0.53
D_x	HMM	60		5	-0.59	-0.59
	LSTM	265	5		-0.53	-0.53
	Hybrid (joint)	275	10	5	-0.53	-0.53
	Hybrid	275	5	5	-0.53	-0.53

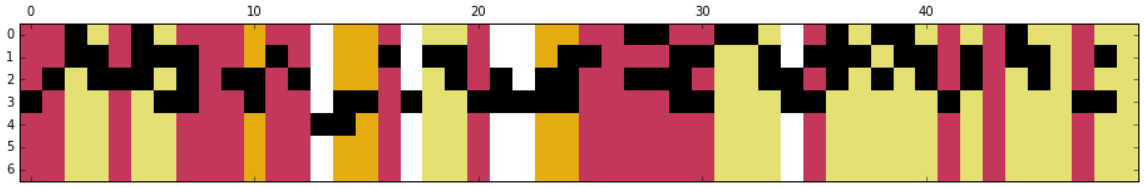
D_s that's easy for the HMM, we set the observation weight vector $\boldsymbol{w_x}$ to 0. To create the data set D_x that's easy for the LSTM, we set the state weight vector $\boldsymbol{w_s}$ to 0. The combination data set D_{sx} uses nonzero values for both weight vectors.

We experimented with different values for the weight parameters a and b, looking for cases where the resulting synthetic data sets have the HMM and LSTM performing differently. If a is too small, for example $a = 2$, then the weak dependence of the outputs on the state is insufficient to give the HMM an advantage over the LSTM. If b is too large, for example $b = 10$, then the strong dependence of the outputs on the observations makes it easy for the HMM to keep up with the LSTM. The HMM and LSTM achieve the same likelihood on these cases, and the hybrid algorithm follows suit. We will focus on a more interesting intermediate case with $a = 5$ and $b = 2$, and investigate how the hybrid algorithm performs there.

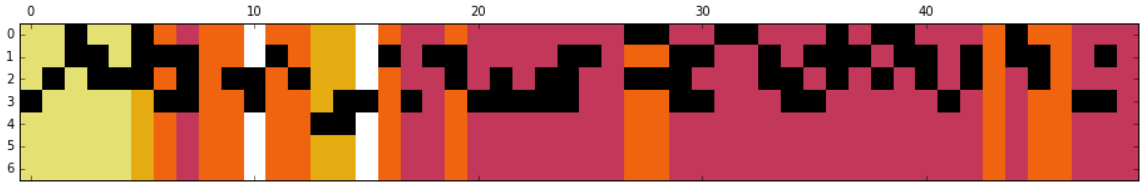
In Table 18, we compare the validation set performance on the synthetic data sets. On the D_s data set designed to favor the HMM, the LSTM does somewhat worse than the HMM and hybrid. On the D_x data set designed to favor the LSTM, the HMM does worse than the LSTM and hybrid. In both cases, the hybrid algorithm performs



(a) HMM states on D_s data set.

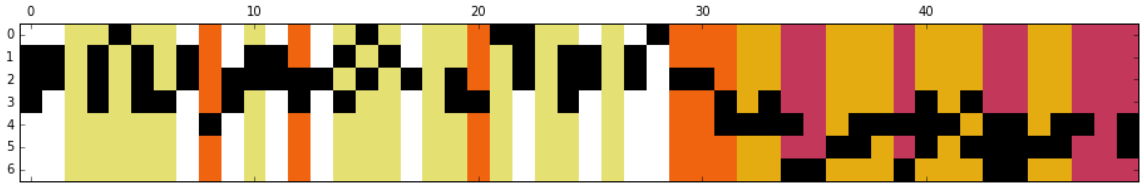


(b) K-means clusters on hybrid LSTM state vectors on D_s data set.

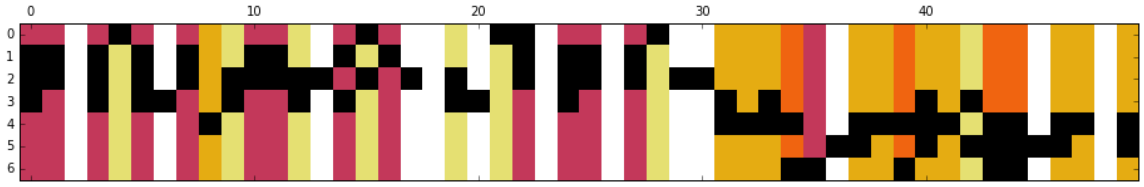


(c) K-means clusters on standalone LSTM state vectors on D_s data set.

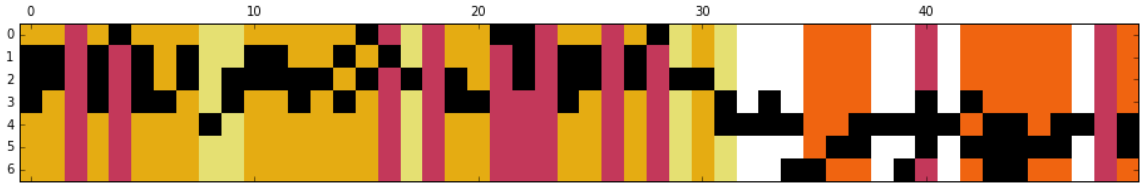
Figure 25: Visualizing the learned model states on the first 50 points of the D_s data.



(a) HMM states on D_{sx} data set.

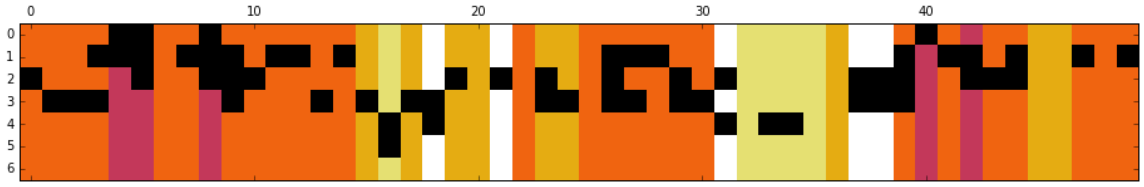


(b) K-means clusters on hybrid LSTM state vectors on D_{sx} data set.



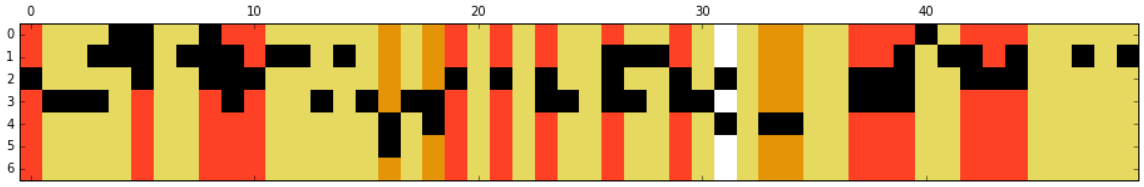
(c) K-means clusters on standalone LSTM state vectors on D_{sx} data set.

Figure 26: Visualizing the learned model states on the first 50 points of the D_{sx} data.



(a) HMM states on D_x data set.

(b) K-means clusters on hybrid LSTM state vectors on D_x data set.



(c) K-means clusters on standalone LSTM state vectors on D_x data set.

Figure 27: Visualizing the learned model states on the first 50 points of the D_x data.

similarly to the better method. On the combination data set D_{sx}, the HMM and LSTM perform similarly and hybrid does better. Overall, the hybrid algorithm performs well in all of these setups, combining the strengths of the HMM and LSTM.

We also tried increasing the LSTM state dimension, which does not seem to affect the performance of the LSTM or the hybrid. The hybrid algorithm has only 10 more parameters than the standalone LSTM - unlike in the text application, the matrix in the linear layer has a small dimension (5x2) due to the binary output.

In Figures 25, 26 and 27, we compare the HMM and LSTM states for the hybrid and the standalone LSTM on the first 50 points in the synthetic data sets. The black cells indicate observations equal to 1. The other cells indicate observations equal to 0, with column colors representing either 5 HMM states or 5 k-means clusters on 5-dimensional LSTM state vectors.

On the D_x data set that favors the HMM, Figure 25 shows that the HMM learns to distinguish between two states in the $(28, 42)$ interval, while the standalone LSTM does not, and the hybrid further refines some of these distinctions, such as between points 34 and 35. On the combination data set D_{sx} in Figure 26, we notice that the first 20 points are likely generated by the first two underlying states, while the last 20 are likely generated by the last two. The LSTM puts some observations generated by different underlying states into the same cluster (e.g. points 5 and 40), while the HMM avoids this. The hybrid makes this error as well, since it only needs to fill in the gaps in the HMM, and picks up on some points with at most one observation equal to 1 (the white cluster). On the LSTM-favoring data set D_x in Figure 27, the LSTM learns that the intervals $(37 - 39)$ and $(42 - 44)$ are similar (red cluster) and so does the hybrid (orange cluster), while the HMM does not. Overall, we can see that the hybrid LSTM fills in the gaps in the HMM.

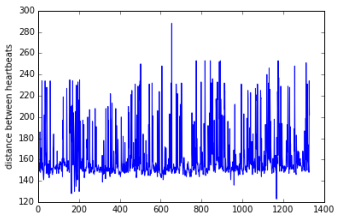
Healthcare time series data

We apply our hybrid algorithm and the LSTM and HMM baselines to ICU time series data from the 2014 Physionet challenge. The objective of the challenge was to use physiological signal features such as ECG measurements to detect heart beats. There are 216,000 time points for each patient. When predicting the location of a heart beat, locations within a 150 ms window of the reference location are considered correct. The 6 features are discretized as binary prior to running the algorithms.

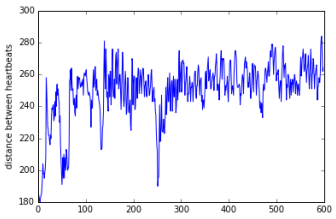
We take several patients from this data set with the most uneven heart beats (Patients 1-3 with patient IDs 1009, 1020 and 1022), as shown in Figure 28. We then predict future heart beats based on past heart beats for the same patient. We also put together a multi-patient data set, with 15 patients in the training set and 5 patients in the validation set, where we predict heart beats for new patients based on the training data.

Table 19: Predictive loglikelihood on the Physionet data sets (sorted by validation set performance).

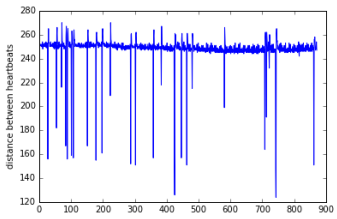
Data	Method	Parameters	LSTM state dimensions	HMM states	Validation	Training
Patient 1	HMM	160		10	-0.675	-0.735
	Hybrid (joint)	670	10	10	-0.41	-0.36
	Hybrid (joint)	245	5	10	-0.4	-0.31
	Hybrid	245	5	10	-0.39	-0.335
	Hybrid	670	10	10	-0.39	-0.335
	LSTM	225	5		-0.37	-0.305
	Hybrid	2120	20	10	-0.36	-0.315
	LSTM	2100	20		-0.343	-0.293
	LSTM	650	10		-0.33	-0.29
Patient 2	HMM	160		10	-0.575	-0.64
	Hybrid (joint)	245	5	10	-0.3	-0.32
	Hybrid (joint)	670	10	10	-0.3	-0.32
	Hybrid	245	5	10	-0.29	-0.315
	Hybrid	670	10	10	-0.29	-0.315
	Hybrid	2120	20	10	-0.28	-0.31
	LSTM	225	5		-0.28	-0.31
	LSTM	650	10		-0.27	-0.303
	LSTM	2100	20		-0.157	-0.217
Patient 3	HMM	160		10	-0.57	-0.59
	Hybrid (joint)	225	5	10	-0.44	-0.43
	Hybrid (joint)	670	10	10	-0.44	-0.43
	LSTM	225	5		-0.42	-0.42
	Hybrid	245	5	10	-0.365	-0.38
	Hybrid	670	10	10	-0.36	-0.375
	Hybrid	2120	20	10	-0.36	-0.375
	LSTM	650	10		-0.337	-0.35
	LSTM	2100	20		-0.333	-0.343
Multi-patient	Hybrid	670	10	10	-0.61	-0.41
	Hybrid	670	10	10	-0.61	-0.44
	HMM	160		10	-0.6	-0.7
	Hybrid	245	5	10	-0.6	-0.46
	Hybrid	245	5	10	-0.57	-0.5
	LSTM	650	10		-0.58	-0.44
	LSTM	225	5		-0.56	-0.51



(a) Distances between heart beats for Patient 1.

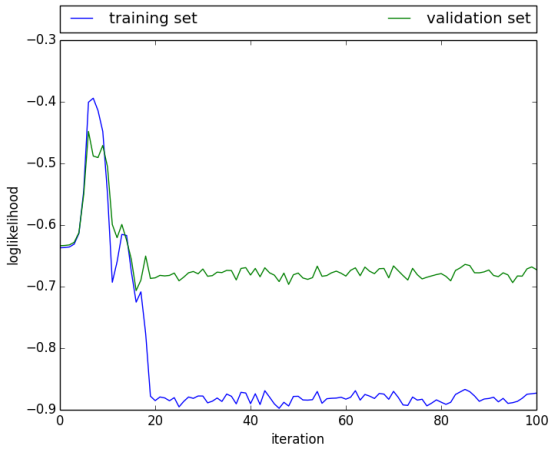


(b) Distances between heart beats for Patient 2.

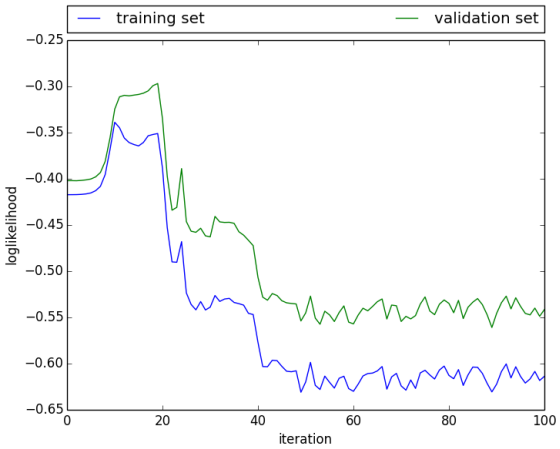


(c) Distances between heart beats for Patient 3.

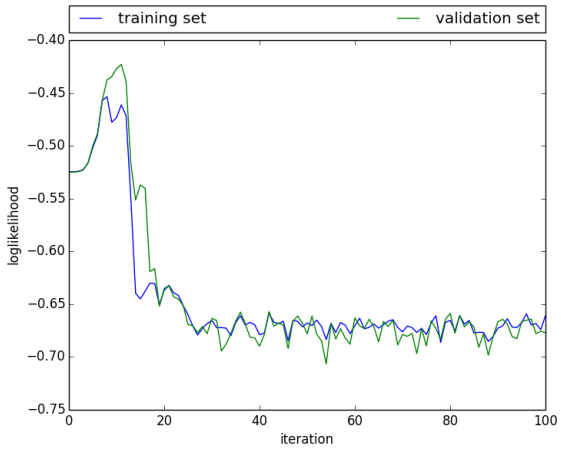
Figure 28: Distances between heart beats in the Physionet data sets.



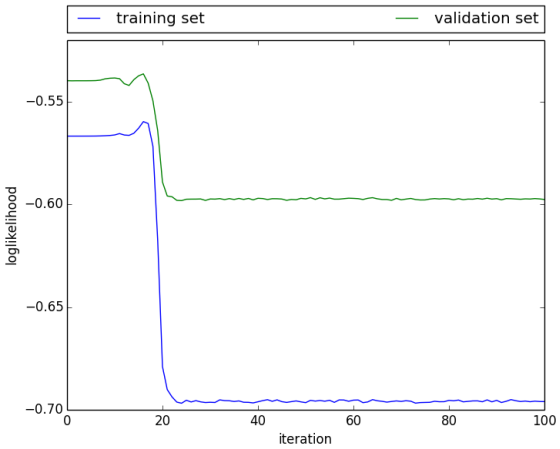
(a) HMM with 10 states for Patient 1.



(b) HMM with 10 states for Patient 2.



(c) HMM with 10 states for Patient 3.



(d) HMM with 10 states for multi-patient data.

Figure 29: Convergence of the predictive loglikelihood for the discrete HMM with 10 states on the Physionet data sets (training set in blue, validation set in green).

Table 19 shows the average performance for the HMM, LSTM and hybrid methods. The HMM convergence plots have suspicious downward curves, as shown in Figure 29, and the HMM performs poorly compared to the other methods on the single-patient data sets, so it's unsurprising that the hybrid does similarly to (or worse) than the LSTM. One possible reason for the HMM's poor performance could be the assumption of independence for the observation features given the states, which might not hold for this data set, as vital sign variables could easily be correlated. On the multi-patient data, however, the HMM performs comparably to the LSTM, but the hybrid methods with 10 LSTM dimensions actually do worse than the HMM. Thus, the HMM performance alone is not sufficient to account for the performance of the hybrid method.

5.4 CONCLUSION

Hybrid HMM-RNN approaches combine the interpretability of HMMs with the predictive power of RNNs. We have some promising preliminary results showing that a small hybrid model can perform somewhat better than a standalone LSTM of the same size. We use visualizations to show how the LSTM and HMM components of the hybrid algorithm complement each other in terms of features learned in the data.

BIBLIOGRAPHY

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. (2016). Concrete Problems in AI Safety. *CoRR*, abs/1606.06565.

Andrei, A. and Kendziorski, C. (2009). An efficient method for identifying statistical interactors in gene association networks. *Biostatistics*, 10(4):706–718.

Andrews, J. L. and McNicholas, P. D. (2014). Variable Selection for Clustering and Classification. *Journal of Classification*, 31(2):136–153.

Bourlard, H. and Morgan, N. (1994). *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers.

Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1):5–32.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *CART: Classification and Regression Trees*. Chapman and Hall.

Che, Z., Purushotham, S., and Liu, Y. (2015). Distilling Knowledge from Deep Networks with Applications to Healthcare Domain. *Neural Information Processing Systems Workshop on Machine Learning for Healthcare (MLHC)*.

Chipman, H. A., George, E. I., and McCulloch, R. E. (2010). BART: Bayesian Additive Regression Trees. *Annals of Applied Statistics*, 4(1):266–298.

Chow, C. K. and Liu, C. N. (1968). Approximating Discrete Probability Distributions with Dependence Trees. *IEEE Transactions on Information Theory*, 14(11):462–467.

Cooper, G. F. (1990). The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artificial Intelligence*, 42(2-3):393–405.

de Campos, C. P., Cuccu, M., Corani, G., and Zaffalon, M. (2014). *Extended Tree Augmented Naive Classifier*, pages 176–189. Springer International Publishing.

Domingos, P. and Pazzani, M. J. (1997). On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. *Machine Learning*, 29:103–130.

Dougherty, J., Kohavi, R., and Sahami, M. (1995). Supervised and Unsupervised Discretization of Continuous Features. In *Machine Learning: Proceedings of the Twelfth International Conference*, pages 194–202. Morgan Kaufmann Publishers, San Francisco,

CA.

Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. New York: John Wiley and Sons.

Evgeniou, T., Pontil, M., and Poggio, T. (2000). Regularization Networks and Support Vector Machines. *Advances in Computational Mathematics*, 13(1):1–50.

Fayyad, U. M. and Irani, K. B. (1993). Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1022–1027. Morgan Kaufmann Publishers, San Francisco, CA.

Friedman, J., Hastie, T., and Tibshirani, R. (2010). Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software*, 33(1):1–22.

Friedman, N., Geiger, D., and Goldszmidt, M. (1997). Bayesian Network Classifiers. *Machine Learning*, 29:131–163.

Gens, R. and Domingos, P. M. (2013). Learning the Structure of Sum-Product Networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML), Atlanta, GA, USA, 16-21 June 2013*, pages 873–880.

Goodman, B. and Flaxman, S. (2016). European Union regulations on algorithmic decision-making and a right to explanation. *2016 ICML Workshop on Human Interpretability in Machine Learning (WHI), New York, NY*.

Grosse, R. B., Salakhutdinov, R., Freeman, W. T., and Tenenbaum, J. B. (2012). Exploiting compositionality to explore a large space of model structures. In *Proceedings of the 28th Conference on Uncertainty in AI (UAI)*, pages 306–315.

Guyon, I., Gunn, S., Ben-Hur, A., and Dror, G. (2005). Result Analysis of the NIPS 2003 Feature Selection Challenge. In *Advances in Neural Information Processing Systems 17*, pages 545–552. MIT Press.

Guyon, I., Gunn, S., Nikravesh, M., and Zadeh, L. A. (2006). *Feature Extraction: Foundations and Applications (Studies in Fuzziness and Soft Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Heckerman, D., Geiger, D., and Chickering, D. M. (1995). Learning Bayesian Networks: The Combination of Knowledge and Statistical Data. *Machine Learning*, 20:197–243.

Hoeting, J., Adrian, D. M., and Volinsky, C. T. (1998). Bayesian Model Averaging. In *Proceedings of the AAAI Workshop on Integrating Multiple Learned Models*, pages 77–83.

AAAI Press.

Jiang, L., Zhang, H., Cai, Z., and Su, J. (2005). Evolutional Naive Bayes. *Proceedings of the 2005 International Symposium on Intelligent Computation and its Applications, ISICA*, pages 344–350.

Karpathy, A., Johnson, J., and Fei-Fei, L. (2016). Visualizing and Understanding Recurrent Networks. *International Conference for Learning Representations Workshop Track*.

Kim, B., Shah, J. A., and Doshi-Velez, F. (2015). Mind the Gap: A Generative Approach to Interpretable Feature Selection and Extraction. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Neural Information Processing Systems (NIPS)*, pages 2260–2268.

Kohavi, R. and John, G. H. (1997). Wrappers for Feature Subset Selection. *Artificial Intelligence*, 97:273–324.

Langley, P. and Sage, S. (1994). Induction of Selective Bayesian Classifiers. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 399–406. Morgan Kaufmann.

Lichman, M. (2013). UCI Machine Learning Repository.

Lipton, Z. C. (2016). The Mythos of Model Interpretability. *2016 ICML Workshop on Human Interpretability in Machine Learning (WHI), New York, NY*.

Maas, A., Le, Q., O'Neil, T., Vinyals, O., Nguyen, P., and Ng, A. (2012). Recurrent Neural Networks for Noise Reduction in Robust ASR. In *Proceedings of INTERSPEECH*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari With Deep Reinforcement Learning. In *Deep Learning Workshop, Neural Information Processing Systems (NIPS)*.

Nguyen, A. M., Yosinski, J., and Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 427–436.

Papernot, N., McDaniel, P. D., Goodfellow, I. J., Jha, S., Celik, Z. B., and Swami, A. (2016). Practical Black-Box Attacks against Deep Learning Systems using Adversarial Examples. *CoRR*, abs/1602.02697.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Poon, H. and Domingos, P. M. (2011). Sum-Product Networks: A New Deep Architecture. In *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*, pages 337–346.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Rao, V. and Teh, Y. W. (2013). Fast MCMC sampling for Markov jump processes and extensions. *Journal of Machine Learning Research*, 14:3207–3232. arXiv:1208.4818.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier. *CoRR*, abs/1602.04938.

Rooshenas, A. and Lowd, D. (2014). Learning Sum-Product Networks with Direct and Indirect Variable Interactions. In Jebara, T. and Xing, E. P., editors, *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 710–718. JMLR Workshop and Conference Proceedings.

Rüping, S. (2006). *Learning interpretable models*. PhD thesis, Dortmund University of Technology.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., and Young, M. (2014). Machine Learning: The High Interest Credit Card of Technical Debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*.

Tang, S., Chen, L., Tsui, K., and Doksum, K. (2014). Nonparametric Variable Selection and Classification: The CATCH Algorithm. *Computational Statistics and Data Analysis*, 72:158–175.

Tibshirani, R. (1996). Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288.

Turner, R. (2016). A Model Explanation System: Latest Updates and Extensions. *2016 ICML Workshop on Human Interpretability in Machine Learning (WHI), New York, NY*.

Vergari, A., Mauro, N. D., and Esposito, F. (2015). Simplifying, Regularizing and Strengthening Sum-Product Network Structure Learning. In *Proceedings (Part II) of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD), Porto, Portugal, September 7-11, 2015*, pages 343–358.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). Show and tell: A neural image caption generator. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 3156–3164.

Webb, G. I., Boughton, J., and Wang, Z. (2005). Not So Naive Bayes: Aggregating One-Dependence Estimators. *Machine Learning*, 58:5–24.

Yosinski, J., Clune, J., Nguyen, A., Fuchs, T., and Lipson, H. (2015). Understanding Neural Networks Through Deep Visualization. In *Deep Learning Workshop, International Conference on Machine Learning (ICML)*.

Zhang, H. (2004). The Optimality of Naive Bayes. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA*, pages 562–567.

Zhang, H., Jiang, L., and Su, J. (2005). Augmenting Naive Bayes for Ranking. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 1020–1027. ACM.

Zhang, Y. and Liu, J. S. (2007). Bayesian Inference of Epistatic Interactions in Case-Control Studies. *Nature Genetics*, 39(9):1167–1173.

Zhao, H., Melibari, M., and Poupart, P. (2015). On the Relationship between Sum-Product Networks and Bayesian Networks. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 116–124.